the Rational edge

e-zine for the rational community

december 2000

COVER STORY

Trends in UML and e-development

Jim Rumbaugh on the new world of e-development, and how the language of visual modeling is meeting the new demands.

enter issue | mission statement | archives

Issue Highlights:

- **Cover Story:** Trends in UML and e-Development
- **News:** More Ways to Achieve Greater Speed, Higher Quality with Rational Suite 2001
- **Rationally Speaking:** Rational Customers Talk about Improving

Editor's Notes

Welcome to the premier issue of *The Rational Edge*, our new monthly, online magazine for the Rational community. Whether you are a manager of a development team, an individual contributor, or the CIO of a growing software company, I think you will find something new and engaging in these Web pages to expand your understanding of e-development.

If you find yourself asking, **"What exactly is e-development?"** then you've come to the right place. Over the coming months, *The Rational Edge* will offer insight and straightforward explanations. You'll see that e-development embraces not only the wide range of software being designed for Web-based businesses. It also includes the software for large and small e-devices which continue to transform our daily lives; it includes all the software and e-infrastructure that runs the Internet itself. And e-development integrates all these activities as today's most challenging realm for the software professional, where the demand for higher quality products is just as great as the demand to get these products to market at lightning speed.

Are you a Code Warrior? If you've already found your place in the wild world of e-development, then you're looking for code samples, tips and tricks, the fast track to guru status with the latest technology from Rational and its partners. Please stay tuned! This inaugural issue will serve as a guide to the various concepts, products, and services that define Rational's mission; by the January issue, we will include more "how-to" content for you.

Rational's mission is to ensure the success of customers who depend on developing or deploying

the Economics of Software Development

software. To support that goal, *The Rational Edge* staff will collect and present the best and brightest ideas from inside and outside Rational Software. If you have articles -- or ideas for articles - that you'd like to submit, click on "Submit an Article" at the right of this page. If you have comments you'd like to share about the contents of this or any upcoming issue of *The Rational Edge*, please email me directly at mperrow@rational.com. I look forward to hearing from you.

Sincerely,

Mike Perrow
Editor

Trends in UML and e-Development

by **Jim Rumbaugh**
Rational Methodologist

The Unified Modeling Language (UML) has gained broad industry acceptance as the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. It simplifies the complex process of software design, making a "blueprint" for construction. The UML definition was led by Rational Software's industry-leading methodologists, Grady Booch, Ivar Jacobson, and Jim Rumbaugh. In the following article, Jim Rumbaugh offers his opinions on the new world of e-development, how its demands are affecting new requirements for the UML, and how these requirements are likely to be met.

The Brave New World

The brave new e-world has turned previous assumptions on their head, and old approaches to business or software will no longer succeed. Recent online startups are worth more than venerable corporations that pound steel. Traditional businesses such as banks and stockbrokers are scrambling to go online to avoid the loss of their customers to new competitors. Music publishers are fighting free electronic distribution of their products. Even the Supreme Court publishes decisions on the Web.

The e-world is now distributed, concurrent, and connected. Distributed, because information is all over the world, in many different places. The day of the monolithic central machine is long over. Concurrent, because activity is decentralized and simultaneous. Neither business decision making nor software programs can live with a single thread of control. Connected, because an action in one place can have profound effects everywhere. A virus maker in the Philippines can bring down half of the servers on the planet. The simple computer systems, languages, and models of the past are inadequate for today's needs.

There are many drivers of this new world. Ten years ago, the Internet was

considered a research toy by most people in industry; now every pizza shop has a Web site (at least in California). More and more companies are entrusting critical operations to enterprise distributed object systems. These object-oriented systems contain distributed servers and databases that are connected to support highly concurrent business operations. More and more industries must interoperate in real time just to do business. Banks, airlines, and telephone companies cannot operate without massive information exchange among all the players in the field. Finally, real-time devices are now ubiquitous -- in cars, appliances, consumer electronics, buildings. Real-time issues are no longer the domain of a specialized few, but the concerns of everyone.

The new e-world brings a lot of power, but that power comes at a high development cost. Concurrent, distributed systems have extremely complex interactions that can be hard to understand, let alone predict. Vague specifications are a major problem. In the past, the specifications for a monolithic system only affected the single system, and if it didn't work exactly as specified, nobody really cared. But now a business system may have to interoperate with another system halfway around the world; both are written by people who have never heard of each other. A failure to follow specifications can introduce errors that propagate around the world. You can't take a snapshot of a distributed system for backup or reboot it if part of it fails. The entire system must keep going in spite of failures, errors, or data corruption of some of its parts. Most systems are now real-time systems. Timing concerns matter a great deal to customers and partners. On top of everything else, performance of complex systems is often nonlinear and cannot be predicted by simple extrapolation.

What can you do about it? You can use the same kinds of approaches available to engineers in every field: modeling before construction, architecture based on experience, a process based on best practices, building with reusable components, and the use of tools to leverage the developer's time and skill. Making these capabilities available to software developers is Rational's business.

The UML in the e-World

Well-suited to the new demands of the brave new e-world, the Unified Modeling Language (UML) was designed to be distributed, concurrent, and connected. It is based on objects. Objects are distributed -- each one maintains its own state, distinct from all others. Objects are concurrent -- each one can potentially execute in parallel with all others. Objects are connected -- each one can send messages to others through a Web of links. UML is not tied to a single platform or programming language; therefore it is well suited to bridge networks of different systems. UML was designed with extensibility in mind, so it can adapt to new issues as they arise.

Development of UML began in 1995 with the combining of the Booch and OMT methods at Rational. We at Rational chose to make it public, and a cooperative effort by a score of companies led to a specification adopted by the Object Management Group (OMG) in 1997. This was UML version 1.1. Rational then gave rights to the UML to the OMG so that UML could serve as a publicly available standard. Since then, an OMG committee with representatives from various companies has worked on clarifying and fixing bugs in the original specification, releasing one update in 1998 (version 1.3) with another one expected by the end of 2000 (version 1.4). Rational experts have been active participants in the committee process. The update process cleaned up many internal problems with the UML metamodel, clarified ambiguities in the original document, improved naming consistency, and fixed a number of features needed in specialized areas. But by and large, most average users will not notice many differences. Probably the biggest change in UML 1.3 was a reformulation of use case relationships, but even that did not represent a very big change. The most important addition in UML 1.4 will be the guidelines for writing profiles -- tailorings of UML for particular application domains. There are lots of specific detailed changes, of course, but overall UML is still the same language with the same capabilities.

UML Extension Efforts by Rational

In parallel with the cleanup work on the UML document, there have been a number of initiatives to extend UML to new application areas, including Web systems and databases. Some of these efforts have been carried out as OMG initiatives, while others have been done by individual companies, such as Rational.

To keep up with the rapidly changing e-business world, almost all companies need to develop Web systems. Work by Jim Conallen and others at Rational has provided a way to model Web systems using UML and Rational Rose. This capability is provided as a UML profile that enables modelers to represent the various kinds of elements that compose a Web application -- client pages, server pages, forms, frames, and so on. The profile contains a set of stereotypes for different elements and their relationships. This approach is described in Conallen's book, *Building Web*

Applications with UML (Addison Wesley Longman, 2000). Recent versions of Rational Rose contain this profile.

Almost all e-business applications need databases. Coordinating programming languages and databases has long been a thorny problem in system development, because each used a different way to declare data structure, leading to subtle inconsistencies and difficulties in moving information among programs and databases. By using a single UML model underlying both programming language code and database schemas, many of these problems can be avoided. Rational has developed a database modeling profile for UML that is supported by versions of Rational Rose. This profile allows a developer to construct a logical model of information and a model of the physical database tables derived from the information. Because there are two models, the database developer can tweak the database structure to optimize it -- an important issue with databases. Because the two models are linked together, changes in one model can be reflected in the other, avoiding the danger of inconsistencies.

OMG Initiatives

OMG initiatives occur through an RFP (request for proposal) process. An OMG task force identifies a need and issues an RFP, stating the requirements and calling for member companies to propose solutions. Several companies normally band together to submit a joint proposal. The OMG encourages submitters of separate proposals to work together to merge their proposals rather than engaging in shoot-outs. This forces compromise, which often leads to some bloating of content, but which also promotes universal adoption, a desirable goal for any standard. For these reasons, the UML is messier than it might have been had it conformed strictly to the views of a single company, but it is more comprehensive and universally adopted.

Several major OMG initiatives include:

Real-time modeling -- One major initiative has been adding real-time modeling capability to UML, an effort led by Bran Selic of Rational. Because of its wide scope, this work has been broken into several smaller RFPs. The first deals with time, scheduling, and performance modeling. An initial proposal has recently been submitted by a consortium containing all the leading players in the real-time modeling area (so it is likely to be accepted). Each initial submission undergoes feedback from the OMG membership and a second pass, so this effort will be completed next year. When that is done, the real-time submitters will take up the second part of the problem: modeling reliability and fault tolerance. Most likely, the same team of submitters will participate in both proposals.

One important aspect of real-time systems is their architectural structure. In 1999, Bran Selic and I developed a UML profile called UML-RT that allows systems to be built hierarchically from encapsulated modules (capsules) with well-defined interfaces (ports) and explicit communications paths (connectors). Since then, we have realized that these architectural concepts are useful for most kinds of systems. At the same time,

preparation for the next major update of UML (version 2.0) has shown that a number of other experts and modeling languages have very similar concepts. For example, the telecom language SDL has similar concepts, as do some hardware description languages. Extending UML to include architectural modeling constructs will therefore be part of the overall UML extension effort and will not be restricted to the real-time group.

Defined execution model -- Perhaps the biggest hole in the original UML specification was the lack of an execution model. The static structure of UML models was precisely defined, but the run-time consequences of these models were vaguely described in words. Omitting the precise specification of run-time behavior was a correct decision originally, because our focus was to get the UML constructs defined and published; however, one of the first RFPs for UML extensions was to define the actions supported by UML and their run-time semantics. Bran Selic and I have participated in this effort for Rational, working with people from companies engaged in real-time modeling, action languages, and telecommunications. All of the participants have joined as a single team. The first version of this proposal has just been submitted. It contains an execution model that supports highly concurrent actions without the overspecification of control necessary in major programming languages. The intent of this proposal is not to invent yet another programming language, but to serve as the semantic base on top of which the effect of programming languages can be precisely defined in UML.

Enterprise computing -- Several initiatives deal with enterprise computing. There are RFPs outstanding on Enterprise Distributed Object Computing (EDOC) and Enterprise Application Integration (EAI). There is considerable overlap among the RFPs (a side effect of OMG democracy) but the submitters are almost the same for both RFPs, and they are taking steps to coordinate the two proposals. These initiatives will define profiles for specifying how to construct large distributed, concurrent, event-driven business systems. Wojtek Kozaczynski is spearheading this work for Rational.

Development process -- Another initiative deals with a framework for specifying software development processes. This would provide a standard way to specify a process, such as RUP (Rational Unified Process). The proponents claim that this would allow organizations using multiple processes to compare them easily. I am skeptical of the value of this -- I think that an organization should pick one process and stick with it -- but, in any case, it is important that RUP be expressible in such a framework, so Rational is participating in this initiative. Philippe Kruchten is the Rational lead on this work.

Other initiatives -- There are also a number of related initiatives, such as a standard for data warehousing, CORBA maps to UML, and the XMI format for the exchange of UML models in text format. The various application domain interest groups are building profiles based on UML, but I won't discuss them because they represent uses of UML, rather than changes to it.

UML 2.0 Work

As a standard, UML is similar in many ways to a programming language. As people use the language, they discover new features that they want to add. It is undesirable to change a language too often, however; users need time to absorb a version of the language and get proficient with it. Also, it takes time to develop tools (such as Rational Rose) to support a language, and changing the language too rapidly risks losing tool support. Nevertheless, new needs do arise and languages do need to evolve to support them. I believe that a major upgrade approximately every five years is appropriate for languages, including UML. Some UML developers have argued for more rapid change, but I believe that they have not adequately weighed the benefits of new features against the costs of instability. In any case, a committee-based political process such as OMG's (or any standards organization) contains a lot of inertia, so the evolution of UML is going to take time no matter what people want.

Planning for a new version of UML has begun. The OMG solicited input on proposed changes, and almost 30 responses were received. Some responses focused on a particular area, and others contained long laundry lists of new features. The responses were condensed and prioritized to obtain a shorter list of changes that could be completed in a reasonable time. The result was a set of RFPs that OMG issued in September 2000:

- *UML infrastructure* -- Reorganize the internal structure of the UML metamodel to be more modular, to better support extensibility, and to better align with other OMG standards, especially the meta-object facility (MOF) and the XMI exchange format. This does not directly support end users, but it simplifies changing the language and writing profiles, which should be cleaner and more usable as a result.
- *OCL* -- The Object Constraint Language (OCL) is a declarative text language for writing constraints. It is used in the UML specification to express the rules for well-formed models. It needs a metamodel of its own (in UML) and additional features to support recent changes to UML. This is a tightly focused initiative and will not affect most end users directly.
- *UML superstructure* -- This is the user-visible stuff. Among the major areas of work are modeling of architectural structure, component-based development, relaxation of restrictions on the structure of activity graphs, and adjustments to some of the UML relationships. Some new notation will probably become available to users. These changes should expand the usability of UML in stages of the development process where it is currently a bit weak.
- *Diagram exchange format* -- The original UML specification defined a metamodel for the semantic model but not for diagram layout. Now the diagram format is needed to permit exchange of diagrams among tools from different vendors. I am hopeful (albeit somewhat doubtful) that this effort will involve only actual tool vendors rather than dilettantes who have never written any tool code. This RFP will be issued later than the others to permit a staggered start.

As UML 2.0 changes are made, an important overriding requirement will

be to maintain compatibility with existing UML models as much as possible. Developers and their companies have made a large investment in their models and don't want to have to change them. In addition, proliferation of new concepts should be discouraged wherever existing concepts can be extended to cover the new needs. There will be an effort to purge unused concepts, however, especially stereotypes with names but no semantics. (Hint: If the definition is one or two sentences with no structure or semantics, it is probably one of these.)

The work will proceed in two stages with intermediate feedback, with a target completion date of January 2002. I fully expect UML 2.0 to slip to the end of the year. (No, I'm not afraid to make predictions in print. Check back in two years to see if I was correct.) I expect a number of multi-company consortia to work on various areas of the problem. There may be several proposals to some of the RFPs, but I expect the submitters to work amicably to merge their proposals in the second stage. The biggest problem will be to restrict feature bloat in the language, which is always difficult in a political process.

Outlook

UML is a usable and useful language today, and you should not wait to begin using it just because it will be extended two years from now. Most of UML will remain unchanged, and most of its extensions will broaden the scope of the language rather than alter existing semantics. Like any language, including the one in which this article is written, UML is a living language that evolves to meet changing needs. Its broad acceptance, coupled with its capacity for enhancement, should give any potential UML user the confidence to make the investment in learning it.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!

Next-Generation Software Economics

by **Walker Royce**
Vice President and General Manager
Strategic Services
Rational Software

Software engineering is dominated by intellectual activities focused on solving problems with immense complexity and numerous unknowns in competing perspectives. The early software approaches of the 1960s and 1970s can best be described as craftsmanship, with each project using a custom process and custom tools. In the 1980s and 1990s, the software industry matured and transitioned to more of an engineering discipline. However, most software projects in this era were still primarily research-intensive, dominated by human creativity and diseconomies of scale. The next generation of software processes is driving toward a more production-intensive approach dominated by automation and economies of scale. Next-generation software economics are already being achieved by some advanced software organizations. Many of the techniques, processes, and methods described in a modern process framework have been practiced for several years. However, a mature, modern process is nowhere near the state of the practice for the average software organization.

This article introduces several provocative hypotheses about the future of software economics. In 1987 Barry Boehm published a one-page description of the "Industrial Software Metrics Top 10 List." This assessment remains a good objective framework for discussing the predominant economic trends of software development. Although many of the metrics are gross generalizations, they accurately describe some of the fundamental economic relationships that resulted from the conventional software process practiced over the past 30 years. Quotations from Boehm's list are presented in italics in the paragraphs below. After each quotation I summarize some of the important economic results of the conventional process and speculate on how a modern

software management framework, or iterative process like the Rational Unified Process (RUP), can improve these performance benchmarks. Although there are not enough project data to validate my assertions, I believe that these expected changes provide a good description of trends a software organization should look for in making the transition to a modern process.

1. *Finding and fixing a software problem after delivery costs 100 times more than finding and fixing the problem in early design phases.*

   This metric dominates the rationale for most software process improvement. Modern processes, component-based development technologies, and architecture frameworks are explicitly targeted at improving this relationship. An architecture-first approach will likely yield ten-fold to hundred-fold improvements in the resolution of architectural errors. Consequently, a modern process places a huge premium on early architecture insight and risk-confronting activities.

2. *You can compress software development schedules 25% of nominal, but no more.*

   An N% reduction in schedule would require an M% increase in personnel resources (assuming that other parameters remain fixed). Any increase in people requires more management overhead. In general, the limit of flexibility in this overhead, along with scheduling concurrent activities, conserving sequential activities, and other resource constraints, is about 25%. This metric should remain valid for the engineering stage of the life cycle, where the intellectual content of the system is evolved. However, if the engineering stage is successful at achieving a consistent baseline, including architecture, construction plans, and feature scope, schedule compression in the production stage should be more flexible. Whether a line-of-business organization is amortizing the engineering stage across multiple projects, or a project organization is amortizing the engineering stage across multiple increments, there should be much more opportunity for concurrent development.

3. *For every $1 you spend on development, you will spend $2 on maintenance.*

   Anyone working in the software industry over the past 10 to 20 years knows that most of the software in operation is considered to be difficult to maintain. A better way to measure this ratio would be the productivity rates between development and maintenance. One interesting aspect of iterative development is that the line between development and maintenance has become much fuzzier. There is no doubt that a mature iterative process and a good architecture can reduce scrap and rework levels considerably. Given the overall homogenization of development and maintenance activities, my guess is that this metric should change to a one-for-one

relationship, where development productivity will be similar to maintenance productivity.

4. *Software development and maintenance costs are primarily a function of the number of source lines of code.*

   This metric is primarily due to the predominance of custom software development, lack of commercial components, and lack of reuse inherent in the era of the conventional process, in which the size of the product was the primary cost driver and the fundamental unit of size was a line of code. Commercial components, reuse, and automatic code generators can seriously pollute the meaning of a source line of code. Construction costs will still be driven by the complexity of the bill of materials. More components, more types of components, more sources of components, and more custom components will result in more integration labor and drive up costs. Fewer components, fewer types, fewer sources, and more industrial-strength tooling will drive down costs. Next-generation cost models should become less sensitive to the number of source lines and more sensitive to the discrete numbers of components and their ease of integration.

5. *Variations among people account for the biggest differences in software productivity.*

   This is certainly a key piece of conventional wisdom: Hire good people. This metric is also a subject of overhype and underhype. When you don't know objectively why you succeeded or failed, the obvious scapegoat is the quality of the people. It is subjective and difficult to challenge. In any engineering venture where intellectual property is the real product, the dominant productivity factors will be personnel skills, teamwork, and motivations. To the extent possible, a modern process emphasizes the need for high-leverage people in the engineering stage, when the team is relatively small.

6. *The overall ratio of software to hardware costs is still growing. In 1955 it was 15:85; in 1985, 85:15.*

   The need for software, its breadth of applications, and its complexity continue to grow almost without limits. The main impact of this metric on software economics is that hardware continues to get cheaper. Processing cycles, storage, and network bandwidth continue to offer new opportunities for automation. Consequently, software environments are playing a much more important role. From a modern process perspective, I can see the environment doing much more of the bookkeeping and analysis activities that were previously done by humans. Configuration control and quality

assurance analyses are already largely automated, and the next frontier is probably significant improvements in automated production and automated testing.

7. *Only about 15% of software development effort is devoted to programming.*

   The amount of programming that goes on in a software development project is probably still roughly 15%. The difference is that modern projects are programming at a much higher level of abstraction. An average staff-month of programming produced maybe 200 machine instructions in the 1960s and 1000 machine instructions in the 1970s and 1980s. Programmer productivity in the 1990s can produce tens of thousands of machine instructions in a single month, even though only a few hundred human-generated source lines may be produced.

8. *Software systems and products typically cost three times as much per instruction as individual software programs. Software-system products (i.e., system of systems) cost nine times as much.*

   This exponential relationship is the essence of what is called "diseconomy of scale." Unlike other commodities, the more software you build, the more expensive per unit item it is. This diseconomy of scale should be greatly relieved with a modern process and modern technologies. Under certain circumstances, such as a software line of business producing discrete, customer-specific software systems with a common architecture, common environment, and common process, an economy of scale should be achievable.

9. *Walkthroughs catch 60% of the errors.*

   This may be true, but walkthroughs are not catching the errors that matter. All defects are not created equal. In general, walkthroughs and other forms of human inspection are good at catching surface issues and style issues; few humans are good at reviewing even first-order semantic issues in a code or model segment. How many programmers get their code to compile the first time? Human inspections and walkthroughs will not expose the significant issues (resource contention, performance bottlenecks, control conflicts, and the like); they will only help resolve them. While the environment catches most of the first-level inconsistencies and errors, the really important architectural issues can only be exposed through demonstration and early testing and resolved through human scrutiny.

10. *80% of the contribution comes from 20% of the contributors.*

This is true across almost any engineering discipline. These are the fundamental postulates that underlie the rationale for a modern software management process framework. 80% of the engineering is consumed by 20% of the requirements. 80% of the software cost is consumed by 20% of the components. 80% of software scrap and rework is caused by 20% of the errors. 80% of the resources are consumed by 20% of the components. 80% of the progress is made by 20% of the people. These relationships are timeless and constitute the background philosophy to be applied throughout the planning and conduct of a modern software management process.

As this discussion illustrates, the next generation of software processes, and specifically the techniques presented in the Rational Unified Process, can help software organizations achieve unprecedented economic advantages. As we head into an era that leverages a more production-intensive approach for software development, dominated by commercial components and automation, we can start to achieve economies of scale.

References

Boehm, B.W. "Industrial Software Metrics Top 10 List," IEEE Software 4, 5 (September 1987), pp. 84-85.

Royce, W. E. *Software Project Management: A Unified Framework*. Addison-Wesley Longman, 1998.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!

The Ten Essentials of RUP

by Leslee Probasco
Development Manager
Rational Unified Process
Rational Software Canada

To effectively apply the Rational Unified Process (affectionately known as "RUP"), it is important to first understand its key objectives, why each is important, and how they work together to help your development team produce a quality product that meets your stakeholders' needs.

Camping Trip? Software Project? Identify Essentials First

The other evening, my neighbor Randy came over to ask for help: He was preparing for a weekend camping and hiking trip and trying to determine what gear to pack. He knows that I often lead and participate in wilderness trips and was impressed with how I'm able to quickly and efficiently determine what items to cram into my limited packing space, while referring to a list of all the equipment and clothing I own. "Do you think I could borrow that list?" he asked.

"Sure, but I'm afraid it won't be much help," I explained. You see, I have literally *hundreds* of items on my outdoor gear list, covering many different types of outings -- from backpacking and climbing, to skiing, snow-shoeing, ice-climbing, and kayaking -- and for trip lengths ranging from simple day trips to multi-day expeditions. I knew that without some guidance, Randy would probably not be able to wade through the multitude of items on my list and figure out what *he* really needed for his relatively simple outing.

Start With Essentials, Then Add the Extras

Instead, I offered to look through the items Randy had already crammed into his bulging pack. We could see what he might eliminate to lighten his load and also whether any necessary items were missing. Within a short time, I could tell that what he really lacked was an understanding of what were the "essentials" for any wilderness outing.

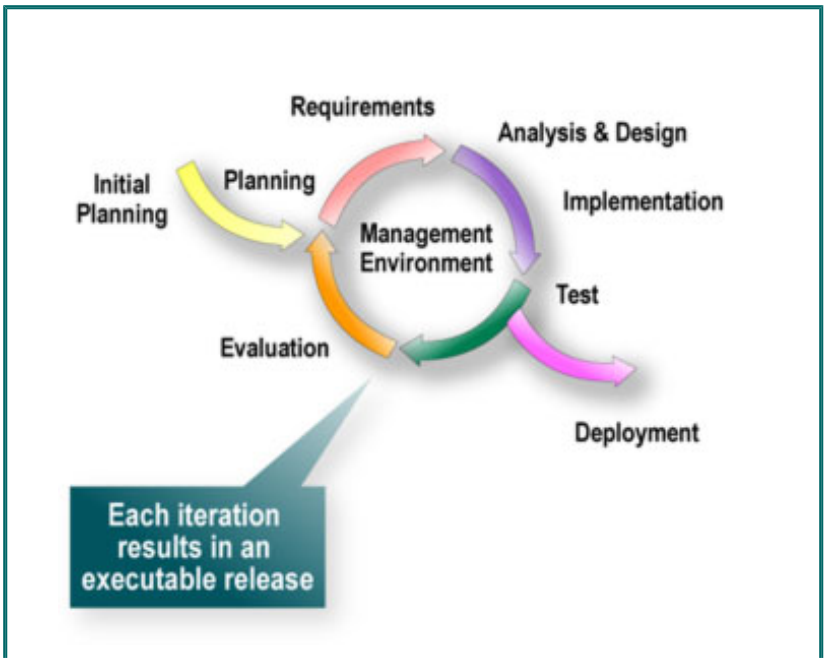
I pulled out a blank sheet of paper and listed ten items[1]:

1. Map
2. Compass
3. Sunglasses & sunscreen
4. Extra clothing
5. Extra food & water
6. Headlamp
7. First-aid kit
8. Fire-starter
9. Matches
10. Knife

"Here, Randy. This is the list you really need. If you start with these 'ten essentials,' then for any trip, the other necessary items will become obvious." I memorized this list many years ago, when I first started mountaineering, and I still refer to it -- no matter what type of trip I am preparing for or how long I'll be gone. Each of these items scales up or down, depending on the trip, but starting with a "short list" and expanding it as needed is much easier than starting from a long list and trying to decide what *not* to take.

Applying This Lesson to RUP

Often, as I help project teams sort through the many elements in RUP, I hear questions such as: "How do I sort through all of these items and determine which ones I need for my project?" "RUP contains so much information. It must be intended only for big projects -- can I really use it for my small one?"



What these folks really need, I've decided, is a "Ten Essentials of RUP," similar to the simple list I gave to my friend Randy -- one that would serve as a reasonable starting point for any project: small, medium, or large. This list would focus on what I call "the

essence" of RUP -- or of any effective software process, for that matter.

For those of you who are unfamiliar with RUP, this diagram represents the incremental and iterative nature of the Rational Unified Process, which covers a vast array of disciplines within the software development lifecycle, including artifacts, guidelines, team member roles, and activities.

Often, projects get bogged down with the details in one particular area before all the participants understand the key process elements required to fully produce and deliver a quality product. Then, when the project falls behind, the blame is placed on some activity that may have been over-emphasized or whose usefulness they don't understand: "See, I told you that requirements management (or use cases, or collecting project metrics, or using configuration management, or using a defect tracking tool, or having status meetings) would slow us down!"

Having an "Essentials" list allows team members to take a more systematic and holistic approach to the overall process of developing software. Once a process framework or "architecture" is in place, *then* team members can more effectively focus on individual problem areas (and often, I'll admit, requirements management is right there at the top of the list). It is also important to identify and prioritize obvious problems and their associated risks at the outset, so the team can apply early mitigation strategies as needed.

The Ten Essentials of RUP

So, what should be on the "Ten Essentials of RUP" list? Here are my choices:

1. Develop a Vision
2. Manage to the Plan
3. Identify and Mitigate Risks
4. Assign and Track Issues
5. Examine the Business Case
6. Design a Component Architecture
7. Incrementally Build and Test the Product
8. Verify and Evaluate Results
9. Manage and Control Changes
10. Provide User Support

Let's look at each of these items individually, see where they fit into RUP, and find out why each made my "short list."

1. **Develop a Vision**

Having a clear vision is key to developing a product that meets your stakeholders' *real* needs.

The Vision captures the "essence" of the *Requirements Workflow* in RUP: analyzing the problem, understanding stakeholder needs, defining the system, and managing the requirements as they change.

The Vision provides a high-level, sometimes contractual, basis for more detailed technical requirements. As the term implies, it is a clear, and usually high-level, view of the software project that can be articulated to any decision maker or implementer during the process. It captures very high-level requirements and design constraints, giving the reader an understanding of the system to be developed. It also provides input for the project-approval process, and is therefore intimately related to the Business Case. And finally, because the Vision communicates the fundamental "why's and what's" of the project, it serves as a means for validating future decisions.

The Vision statement should answer the following questions, which can also be broken out as separate, more detailed items:

- What are the key terms? (Glossary)
- What problem are we trying to solve? (Problem Statement)
- Who are the stakeholders? Who are the users? What are their respective needs?
- What are the product features?
- What are the functional requirements? (Use Cases)
- What are the non-functional requirements?
- What are the design constraints?

2. **Manage to the Plan**

"The product is only as good as the plan for the product."[2]

In RUP, the Software Development Plan (SDP) aggregates all information required to manage the project and may encompass a number of separate items developed during the Inception phase. It must be maintained and updated throughout the project.

The SDP defines the project schedule (including Project Plan and Iteration Plan) and resource needs (Resources and Tools), and is used to track progress against the schedule. It also guides planning for other process components: Project Organization, Requirements Management Plan, Configuration Management Plan, Problem

Resolution Plan, QA Plan, Test Plan, Test Cases, Evaluation Plan, and Product Acceptance Plan.

In a simple project, statements for these plans may consist of only one or two sentences. A Configuration Management Plan, for example, may simply state: "At the end of each day, the contents of the project directory structure will be zipped, copied onto a dated, labeled zip disk, marked with a version number, and placed in the central filing cabinet."

The format of the Software Development Plan is not as important as the activity and thought that go into producing it. As Dwight D. Eisenhower said, "The plan is nothing; the planning is everything."

"Manage to the Plan" -- together with essentials #3, #4, #5, and #8 in our list above -- captures the essence of the *Project Management Workflow* in RUP, which involves conceiving the project, evaluating scope and risk, monitoring and controlling the project, and planning for and evaluating each iteration and phase.

3. **Identify and Mitigate Risks**

An essential precept of RUP is to identify and attack the highest risk items early in the project. Each risk the project team identifies should have a corresponding mitigation plan. The risk list should serve both as a planning tool for project activities and as the basis for specifying iterations.

4. **Assign and Track Issues**

Continuous analysis of objective data derived directly from ongoing activities and evolving product configurations is important in any project. In RUP, regular status assessments provide the mechanism for addressing, communicating, and resolving management issues, technical issues, and project risks. Once the appropriate team has identified the hurdles, they assign all of these issues a due date and a person with responsibility for resolving them. Progress should be tracked regularly, and updates should be issued as necessary.

These project "snapshots" highlight issues requiring management attention. While the period may vary, regular assessment enables managers to capture project history and remove any roadblocks or bottlenecks that restrict progress.

5. **Examine the Business Case**

The Business Case provides the necessary information, from a business standpoint, to determine whether the project is a worthwhile investment. The Business Case also helps develop an economic plan for realizing the project Vision. It provides justification for the project and establishes economic constraints. As the project proceeds, analysts use the Business Case to accurately assess return on investment (ROI).

Rather than delve deeply into the specifics of a problem, the Business Case should create a brief but compelling justification for the product that all project team members can easily understand and remember. At critical milestones, managers should return to the Business Case to measure actual costs and returns against projections and decide whether to continue the project.

6. **Design a Component Architecture**

In the Rational Unified Process, a software system's architecture (at a given point in time) is defined as the organization or structure of the system's significant components interacting, through interfaces, with components composed of successively smaller components and interfaces. What are the main pieces? And how do they fit together?

RUP provides a methodical, systematic way to design, develop, and validate such an architecture. The steps involved in the *Analysis and Design Workflow* include defining a candidate architecture, refining the architecture, analyzing behavior, and designing components of the system.

To speak and reason about software architecture, you must first create an architectural representation that describes important aspects of the architecture. In RUP, this is captured in the Software Architecture Document, which presents multiple views of the architecture. Each view addresses a set of concerns specific to a set of stakeholders in the development process: end users, designers, managers, system engineers, system administrators, and so on. The document enables system architects and other project team members to communicate effectively about architecturally significant project decisions.

7. **Incrementally Build and Test the Product**

The essence of the *Implementation* and *Test* workflows in RUP is to incrementally code, build, and test system components throughout the project lifecycle, producing executable releases at the end of each iteration after inception.

At the end of the elaboration phase, an architectural prototype is

available for evaluation; this might also include a user-interface prototype, if necessary. Then throughout each iteration of the construction phase, components are integrated into executable, tested builds that evolve into the final product. Also key to this essential process element are ongoing Configuration Management and review activities.

8. **Verify and Evaluate Results**

As the name implies, the Iteration Assessment in RUP captures the results of an iteration. It determines to what degree the iteration met the evaluation criteria, including lessons learned and process changes to be implemented.

Depending on the scope and risk of the project and the nature of the iteration, the assessment ranges from a simple record of a demonstration and its outcomes to a complete, formal test review record.

The key here is to focus on process problems as well as product problems. The sooner you fall behind, the more time you will have to catch up.

9. **Manage and Control Changes**

The "essence" of RUP's *Configuration and Change Management Workflow* is to manage and control the scope of the project as changes occur throughout the project lifecycle. The goal is to consider all stakeholder needs and meet them to whatever extent possible, while still delivering a quality product, on time.

As soon as users get the first prototype of a product (and often even before that!), they *will* request changes. It is important that these changes be proposed and managed through a consistent process.

In RUP, Change Requests are used to document and track defects, enhancement requests, and any other type of request for a change to the product. They provide an instrument for assessing the impact of a potential change as well as a record of decisions made about that change. They also help ensure that all project team members understand the potential impact of a proposed change.

10. **Provide User Support**

In RUP, the "essence" of the *Deployment Workflow* is to wrap up and deliver the product, along with whatever materials are necessary to assist the end-user in learning, using, and maintaining the product.

At a minimum, a project should supply users with a User's Guide -- perhaps implemented through online Help -- and possibly an Installation Guide and Release Notes. Depending on the complexity of the product, users may also need training materials. Finally, a Bill of Materials clearly documents what should be shipped with the product.

What about Requirements?

Some of you may look at my list of essentials and vehemently disagree with my choices. You may ask, for example, where "requirements" fit into this picture. Aren't they essential? I'll tell you why I have not included them on my list. Sometimes I'll ask a project team (especially a team for an internal project), "What are your requirements?" and receive the response, "We don't really have any requirements."

This amazed me at first (since I come from a military-aerospace development background). How could they not have any requirements? As I talked to these teams further, I found out that to them, "requirements" meant a set of externally imposed "shall" statements about what they must have or the project will be rejected -- and they really don't have any of those! Especially if a team is involved in research and development, the product requirements may evolve throughout the project.

So for these projects, I follow up their response with another question: "Okay, then what is the vision for the product?" Then their eyes light up. We move easily through each of the questions listed as bullet points under RUP essential #1 above ("Develop a Vision"), and the requirements just flow naturally.

For teams working on contracts with specified requirements, it may be useful to have "Meet Requirements" on their essentials list. Remember, my list is meant only as a starting point for further discussion.

Summary: Applying the Ten Essentials

So, how can discovery of the "Ten Essentials of RUP" make a difference in my life? Here are a few ways that these recommendations can help me work with projects of varying sizes.

For Very Small Projects

First of all, if someone asks me how they might use RUP and the Rational development tools for building a simple product with a very small, inexperienced team that is just learning about process, I can share my "Ten Essentials" list and avoid overwhelming the project team with all the details in RUP and the full power of the Rational Suites of tools.

In fact, these ten essentials can be implemented without *any* automated tool support! A project notebook with one section devoted to each of the ten essentials is actually a very good starting point for managing a small project. (And I have found Post-It Notes invaluable for managing, prioritizing, and tracking change requests on small projects!)

For Growing Projects

Of course, as a project's size and complexity grow, these simple means of applying the ten essentials soon become unmanageable, and the need for automated tools will become more obvious. Nevertheless, I would still encourage team leaders to start with the "Ten Essentials" and "Best Practices" of RUP and incrementally add tool support as needed, rather than immediately attempt to fully utilize the complete set of tools in the Rational Suites.

For Mature Project Teams

For more mature project teams that may already be applying a software process and using development tools, the "Ten Essentials" can help provide a quick method for assessing the balance of key process elements and identify and prioritize areas for improvement.

For All Projects

Of course, each project is different, and it may seem that some projects don't really need all of these "essentials." In these cases, it is also important to consider what will happen if your team ignores any of these essentials. For example:

- No vision? You may lose track of where you are going and wind up taking unproductive detours.
- No plan? You will not be able to track progress.
- No risk list? Your project is in danger of focusing on the wrong issues now and may get blown up by an undetected "land mine" five months from now.
- No issues list? Without timely analysis and problem solving, small issues often evolve into major roadblocks.
- No business case? You risk losing time and money on the project. Eventually it may run out of funds and be cancelled.
- No architecture? You may be unable to handle communication, synchronization, and data access issues as they arise. You may also have problems with scaling and performance.
- No product (prototype)? You won't be able to test effectively and will also lose credibility with customers.
- No evaluation? You'll have no way of knowing how close you really are to meeting your deadlines, project goals, and budget.
- No change requests? You'll have no way to assess the potential impact of changes, prioritize competing demands, and update the

whole team when changes are implemented.

- No user support? Users will not be able to use the product to best advantage, and tech support may be overwhelmed with requests for help.

So, there you have it -- it's very risky to live without knowledge of the "Ten Essentials." I encourage you to use these as a starting point for your project group. Decide what you want to add, change, or take away. Then, decide what else is really "essential" for your project -- no matter what size it may be -- to deliver a product on time, within budget, that meets your stakeholders' *real* needs!

Other Essentials

Other organizations have published similar lists of software project essentials. IEEE Software Magazine, March/April 1997, included an article by Steve McConnell, "Software's Ten Essentials." The Software Project Manager's Network includes a listing of "**16 Critical Software Practices,**" available at www.spmn.com. And the Software Engineering Institute's (SEI) Capability Maturity Model (CMM) contains Key Process Areas (KPAs) which might also be considered "essentials" (see www.sei.cmu.edu).

---

[1]For a complete analysis of this "Ten Essentials" list, see pages 35-41 of *Mountaineering: The Freedom of the Hills*, 6th edition, published by The Mountaineers of Seattle, WA in 1997.

[2]From the *Johnson Space Center Shuttle Software Group*. Quoted in "They Write the Right Stuff," by Charles Fishman, Fast Company, Issue 6, p. 95, December 1996.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!

From Craft to Science: Searching for First Principles of Software Development

by **Koni Buhrer**
Software Engineering Specialist
Rational Software

Developing large software systems is notoriously difficult and unpredictable. Software projects often are canceled, finish late and over budget, or yield low-quality results -- setting software engineering apart from established engineering disciplines. While puzzling at first glance, the shortcomings of software "engineering" can easily be explained by the fact that software development is a craft and not an engineering discipline. To become an engineering discipline, software development must undergo a paradigm shift away from trial and error toward a set of first principles.

*In this first article of a two-part series for **The Rational Edge**, I will provide support for this theory and propose a first principle of software development, along with a corresponding set of "axiomatic" software requirements. In my next article, I will outline a set of design rules -- the "universal design pattern" -- featuring four types of design elements that, in combination, can describe an entire software system in a single view.*

Crafting vs. Engineering

When I look at the fabulous Gothic cathedrals in Europe, I often wonder: Who were the architects that designed and constructed them? How did they acquire the knowledge to create structures that were not only beautiful, but also sturdy enough to withstand the forces of nature for centuries? How were they able to construct these buildings without computers, hydraulic tools, or modern construction materials?

Clearly, the medieval architects had extraordinary knowledge of building design and construction -- knowledge for which they were highly respected and honored as masters of their trade. And masters they were; but they do not deserve to be called engineers.

Why not? The reason is simple. While the medieval architects had profound knowledge of how to construct tall buildings, they knew nothing about the laws of physics. They knew that the shape of an arched ceiling had to be conical, but they had never heard of differential equations. The knowledge of the medieval architects was based exclusively and entirely on experience from trial and error and did not have any scientific foundation. In a nutshell: The medieval architects knew *how* to do things, but they had no idea *why* things had to be done that way[1]. Building design and construction was a craft, and the medieval masters of architecture were craftsmen and not engineers.

The difference between the medieval master of architecture and the modern construction engineer is that the engineer understands the *reasons* for the architectural rules. He can deduce the rules from the laws of physics and therefore prove the soundness of a design before doing any construction work, even if he has no applicable experience. The medieval architect, by contrast, had to accumulate his knowledge slowly and painfully through trial and error. Eventually he would become a master and pass on this knowledge to his apprentices. Some of the apprentices would in time become masters themselves and pass on their knowledge to their apprentices. And so the medieval craft of architecture slowly matured over time -- all the while remaining a craft.

Building design and construction abruptly advanced from a craft to an engineering discipline when it was based upon the scientific foundation provided by the laws of physics. Maturity alone does not turn a craft into an engineering discipline; engineering requires a paradigm shift. Crafting is based on trial and error, while engineering is based on a set of first principles (often the applicable laws of physics) from which all knowledge can be inferred. Trial and error is the hallmark of crafting; first principles are the hallmark of engineering.

What Is Software Development Today?

With this distinction in mind, let's ask ourselves: Is software development today a craft, an engineering discipline, or something in between? Many software developers would probably assert that software development is not yet an established engineering discipline, but it is well on its way to becoming one.

I think that is a delusion. In my view, software development is pure craft. It can't be an engineering discipline -- and in fact does not embody any aspect of engineering -- because it has no first principles. All knowledge on how to develop software is based exclusively on trial and error. Yes, we have made some progress since the dawn of software development. Masters of software development like Grady Booch, Jim Rumbaugh, and Walker Royce, for example, have invented successful methods and rules, often called *best practices*, to guide the software developer. Yet, like the

medieval architects, software developers have learned and validated those best practices through trial and error. What software development lacks at this time is a set of first principles on which its rules and methods could be based[2].

The lack of first principles[3] of software development has serious consequences, often captured by the phrase "the software crisis." Software projects are often canceled before completion, and the projects that *are* completed are often over budget, late, and yield low-quality results. As others have observed, the key factor for the success of a software project is a good architecture. And the inability to routinely create a good architecture is exactly what distinguishes software development from established engineering disciplines. Why this embarrassing discrepancy? Well, unlike the software developer, the engineer uses a set of first principles to prove the adequacy of a design before any construction work is done. The software developer, on the other hand, must rely on testing to assess the quality of an architecture. Or, to put it plainly, the software developer has to find a good architecture by trial and error.

And that explains why the two most widely seen problems in unsuccessful software projects are "late scrap and rework" and "late design breakage." The software developer creates an architectural design early during software development, but has no means to immediately assess its quality. The software developer has no first principles at hand to prove the adequacy of a design. Testing the software eventually reveals all architectural defects, but only in a late development phase when fixing them is costly and disruptive.

What Is Different about Software Engineering?

The inability of the software developer community to turn software development into a successful engineering discipline has caused quite a bit of embarrassment. The low success rate of software projects clearly sets software development apart from established engineering disciplines. Many reasons have been proposed to explain why software *engineering* should be different from other engineering disciplines. Let's look at three well-known examples.

Every engineering discipline involves elements of art and science. This is true, but it is the scientific element -- not the artistic element -- that distinguishes engineering from craft. And the scientific element is exactly what software development is lacking, because it has no first principles on which any science could be based. Moreover, the artistic element of engineering is negotiable. A bridge designed by a civil engineer may be ugly, but it will always be safe. Software development, however, is all art (or craft), and the quality of a software system depends solely on the artistic capabilities of a software developer[4].

Other engineering disciplines have a long history and lots of past

experience, but software engineering is still in its fledgling state.
Many engineering disciplines do indeed have a long history, because they emerged from an ancient craft. But the transition from a craft to an engineering discipline is not a matter of maturity; it is always the result of a paradigm shift away from trial and error toward a set of first principles. Most modern engineering disciplines underwent this paradigm shift quite abruptly when they became based upon the laws of physics. Software development, however -- because it lacks any scientific foundation -- is still a craft. Although it will mature over time, maturity alone will not turn it into an engineering discipline.

Software projects are so unpredictable because software has unlimited flexibility.
This is nonsense. The truth is that many software developers neither know how to create good software architectures nor know how to spot the bad ones. The lack of first principles in software development makes it impossible to objectively distinguish between good and bad architectures. And as a result, software architectures are often bad. Yet the great flexibility of software -- and it is true that software is very, very flexible -- allows a project to progress a long way toward implementing even the worst architecture. Eventually, however, architectural defects become major obstacles that cause extensive scrap and rework, time and budget overruns, and generally poor software quality. So, it is not the great flexibility of the software that makes software projects hard to manage; it is the inability of software developers to routinely and predictably create good architectures.

Software Development Is Inherently a Design Activity

How did we get into this mess? Why does the spirit of trial and error so deeply permeate software development? Why haven't we explored the fundamental laws of software development and discovered its first principles yet?

To answer these questions, we need to understand that software development is inherently a *design* activity with no aspect of *construction* or *manufacturing*. You may find this claim hard to swallow, but it can easily be justified. We have a good understanding of what design is, where it ends, and where construction or manufacturing starts. Consider the following two arguments:

1. The boundary between design

Puzzled by Software Development Issues? There's a Simple Explanation!

The fact that software development is inherently a design activity helps to explain a number of otherwise puzzling issues. Here are some examples:

Q. Why is it much harder to create a detailed work breakdown structure for software implementation than for skyscraper construction?

A. Because software implementation is the equivalent of skyscraper design (i.e., the creation of the

and construction is always clearly marked by an artifact: the blueprint. Design encompasses all the activities needed to create the blueprint; construction encompasses all the activities needed to create products from the blueprint. In a perfect world, the blueprint would specify the product to be created in every detail -- which of course is rarely the case. Still, the purpose of a blueprint is to describe the product to be constructed as precisely as possible. Does the architectural design of a software system describe the software product "as precisely as possible"? No way. The architectural design is intended to describe the essentials, but certainly not all the details of a software system. So the architectural design is clearly not a blueprint. Only the high-level language code describes all the details of a software system, and thus qualifies as the software's blueprint. And because all activities leading up to the blueprint are design, all software development must be design.

2. The effort (time, money, resources) needed to create a commercial product can always be divided into design effort and manufacturing effort. What is the difference? The design effort is common to all copies of the product and has to be expended only once. The manufacturing effort has to be expended every time a copy of the product is created. A software product is typically the binary executable of a program delivered on a CD-ROM. Clearly, the effort to create the source code of a program -- including architectural design, detailed blueprints for a skyscraper) and not skyscraper construction. Design work is naturally less amenable to planning than construction work -- which is true for skyscrapers as well as for software -- because the scope and complexity of the end product are discovered only in the course of the design work.

Q. Why can we implement two software modules in parallel, whereas the floors of a building have to be constructed sequentially?

A. Because the equivalent of implementing a software module is designing -- and not constructing -- a building floor. And two architects can very well design two building floors in parallel, as long as they observe certain interface constraints -- just as two software developers can implement two modules in parallel. The compiler, on the other hand -- which truly constructs the software product -- must compile the modules in the right sequence, just as building floors have to be constructed in the right sequence.

Q. Why can't we achieve economies of scale in software development?

A. The diseconomy of scale we see in software development is inherent to its design nature. We know from other industries that economies of scale apply only to manufacturing processes but not to design tasks. For good reason: Design not only has to deal with each of the design elements, but also with all interactions between the design elements.

> design, and high-level language code -- has to be expended only once, no matter how many copies of the software are produced. Consequently, the effort to create the source code of a program is entirely a design effort, and all software development is design.

Software developers do not construct software; they *design* software. The final result of that design effort -- the high-level language code[5] -- is the blueprint of the software. And it is the compiler/linker that mechanically *constructs* the software product -- a binary executable -- from the high-level language code. The architectural design of a software system most closely corresponds to the cardboard models or design sketches used in some engineering disciplines.

Now, to understand why software developers have not sought and found first principles yet, let's imagine a world where creating a skyscraper would require no more than a detailed blueprint. With the blueprint, an architect could, at the push of a button, have the skyscraper constructed -- instantly, and at virtually no cost. The architect would then test the skyscraper and compare it to the specifications. Should it collapse or not pass the test, the architect could have it demolished and the rubble removed -- instantly, and again at no cost. Would this architect spend much time on formally verifying that the skyscraper design complies with the laws of physics? Or even try to explore and understand those laws? Hardly. He could probably get results more quickly by constructing, testing, and demolishing the skyscraper a couple of times, while fixing the blueprint each time around. In a world where construction and demolition are free, The number of interactions between design elements rises with the number of elements in a non-linear fashion. This is true even with extensive reuse and commercial off-the-shelf components. Software development, which is pure design, will therefore never achieve true economies of scale.

Q. Can we profit from economies of scale in software *manufacturing*?

A. No, because software products are manufactured by the compiler, the linker, and other operating system tools at a largely negligible cost. The sales price of a software product thus only has to cover its design cost, but practically no manufacturing cost. In other industries it is the manufacturing cost of each copy of the product that substantially decreases if more copies are produced. This does not apply to software products.

Q. Why does software have such a low level of reuse?

A. In most industries, reusing existing design elements has a double benefit. It speeds up system design and decreases manufacturing cost. Electronics components, for example, are so popular because they are cheap -- a result of mass production. Using off-the-shelf electronics components thus lowers the traditionally high manufacturing cost of electronics systems. Unfortunately that doesn't apply to software, because software is manufactured at a largely negligible cost already.

trial and error is the approach of choice, and basic research is for suckers.

> Integrating off-the-shelf components doesn't save on manufacturing; on the contrary, it may incur royalty fees.

Well, software development happens in exactly such a world. The software developer creates a blueprint of the software in the form of a high-level language program. He then lets the compiler and the linker construct the software product in the blink of an eye, at virtually no cost. Yes, creating the blueprint requires considerable effort, but construction by the compiler and the linker is practically free. And the software developer certainly doesn't worry about demolition and removal of debris -- at least not until he runs out of disk space. No wonder that the trial and error spirit is deeply entrenched in the software development process, and that the software developer community has not bothered to explore the first principles of software development.

In fact, trial and error has brought us a long way. But with the increasing complexity of modern software systems we are approaching a hard limit. Beyond a certain level of complexity it becomes impossible to create quality architectures by trial and error[6].

Proposal: A First Principle of Software Development

Without first principles, software development will remain a craft forever, and the software crisis (cancelled projects, or results that are over-budget, late, and of poor quality) will persist. So, let's go out and find some first principles -- but where? The laws of physics don't apply to software. Does that mean first principles of software development do not exist? Or have we just not looked carefully enough to find them?

Let's consider how the presence of a first principle -- say, the law of gravity, for instance -- affects the architecture of a system. From an abstract viewpoint, a first principle does nothing more than induce non-negotiable requirements, which the architect has to deal with. I will call those requirements *axiomatic* requirements. For example, in construction a simple axiomatic requirement would be, "The building shall withstand the force of gravity." What makes axiomatic requirements so special is that they apply to each and every system that ever has been and ever will be constructed. Axiomatic requirements are in fact so obvious and common that they are usually implicit. But from the axiomatic requirements, established engineering disciplines have derived a set of architectural rules -- rules all designs must comply with to be worthy of consideration. Designs that violate these rules are obviously at odds with the first principles and are probably faulty.

If it is too hard to find the first principles of software development, it is easy enough to find some axiomatic requirements -- requirements that apply to each and every software system. Here is the list I'd like to propose:

1. The software must obtain input data from one or more external (hardware) interfaces.
2. The software must deliver output data to one or more external (hardware) interfaces.
3. The software must maintain internal data to be used and updated on every execution cycle.
4. The software must transform the input data into the output data (possibly using the internal data).
5. The software must perform the data transformation as quickly as possible.

That sounds brain-dead, doesn't it? But believe me: Very few software systems have an architecture that *obviously* satisfies the axiomatic requirements above. Many software systems may implicitly satisfy them, but not obviously![7]

The axiomatic requirements above seem to be related to the hardware environment of a software system to some degree. I would therefore postulate that the underlying first principle of software development is: "Software runs on and interacts with hardware -- hardware that has only finite speed."

That the hardware environment should play such a dominant role in software architecture is a little surprising. Many software design approaches seem to ignore the hardware environment in the logical, structural, and dynamic views of the architecture. Hardware is often treated as a deployment issue with little significance for architectural structure and behavior. I, however, believe that the hardware environment should be a primary driving force for the architectural design of a software system.

Next Step: Defining a Universal Design Pattern

In my next article for *The Rational Edge*, slated for the January issue, I will focus on the design rules associated with the first principle and its axiomatic requirements. These rules represent a highly useful, universal design pattern to guide software design. The universal design pattern is a first step toward making software development a predictable, repeatable engineering activity -- so don't forget to come back!

[1]More precisely, the medieval master had no means of formally justifying it. For example, if an apprentice would ask the master, "Why do I have to do it like this?" the master would answer, "Because I said so!" A modern engineer, by contrast, would answer, "Well, according to the laws of static mechanics. . . if we take this formula. . . " and so on, thus providing a formal justification of why things have to be done in a certain way.

[2]Some might argue that the best practices constitute the first principles of software development. But unfortunately that is not the case. The reason is that we do not know whether those best practices are actually good practices. We don't have any understanding of *why* a software development practice deserves to be called good or bad -- other than that it proved to be successful in some past application. Identifying best practices by trial and error might eventually lead to deeper understanding of the first principles and mechanisms that determine the quality of a software development practice. But by itself, identifying best practices does not advance software development past its current stage of a craft.

[3]To be clear, first principles are **not** guiding principles. First principles are well-defined axioms that allow formal verification of software development rules and procedures.

[4]Let me elaborate. If asked whether a design was sound, an engineer might answer, "I applied the formulas of static mechanics to verify it." A software developer, however, would answer, "I'm confident, because it looks good" -- much like an artist would judge a sculpture.

[5]Or the detailed design if the compiler can generate code directly from the design models.

[6]Iterative development only expands the limit of complexity that can be mastered by trial and error.

[7]A popular design approach is to identify nouns in the requirements documents and associate a class with each noun. System functionality is modeled by messages the classes exchange. This design approach invariably results in software architectures that make verifying the axiomatic requirements very difficult.

From Waterfall to Iterative Development -- A Challenging Transition for Project Managers

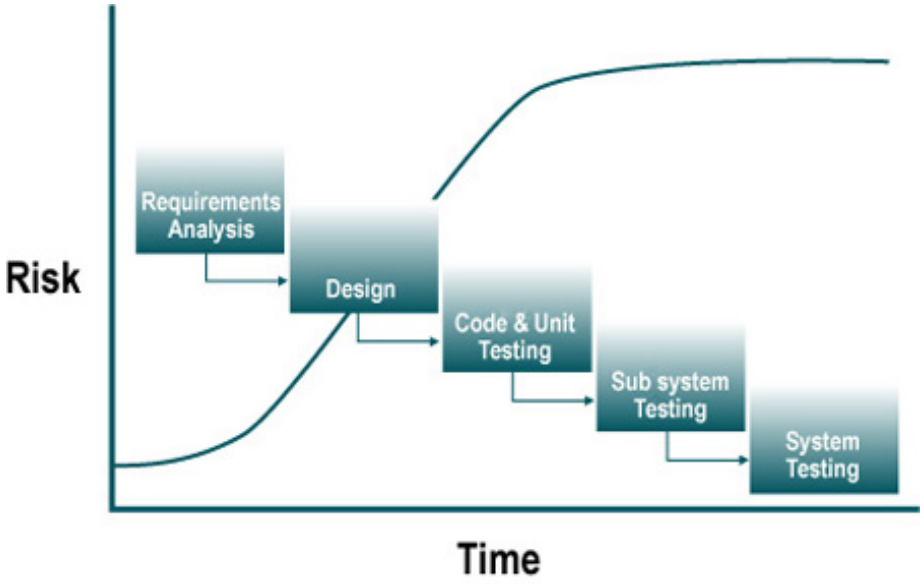
by **Philippe Kruchten**
Rational Fellow

The Rational Unified Process (RUP) advocates an iterative or spiral approach to the software development lifecycle, as this approach has again and again proven to be superior to the waterfall approach in many respects. But do not believe for one second that the many benefits an iterative lifecycle provides come for free. Iterative development is not a magic wand that when waved solves all possible problems or difficulties in software development. Projects are not easier to set up, to plan, or to control just because they are iterative. The project manager will actually have a more challenging task, especially during his or her first iterative project, and most certainly during the early iterations of that project, when risks are high and early failure possible. In this article, I describe some of the challenges of iterative development from the perspective of the project manager. I also describe some of the common "traps" or pitfalls that we, at Rational, have seen project managers fall into through our consulting experience, or from reports and war stories from our Rational colleagues.

Iterative Development

Classic software development processes follow the waterfall lifecycle, as illustrated in Figure 1. In this approach, development proceeds linearly from requirements analysis through design, code and unit testing, subsystem testing, and system testing, with limited feedback on the results of the previous phases.

Figure 1: The Waterfall Development Process



The fundamental problem of this approach is that it pushes risk forward in time, where it's costly to undo mistakes from earlier phases. An initial design will likely be flawed with respect to its key requirements, and furthermore, the late discovery of design defects tends to result in costly overruns and/or project cancellation. The waterfall approach tends to mask the real risks to a project until it is too late to do anything meaningful about them.

An alternative to the waterfall approach is the iterative and incremental process, as shown in Figure 2.

In this approach, built upon the work of Barry Boehm's spiral model (see "Further Reading"), the identification of risks to a project is forced early in the lifecycle, when it's possible to attack and react to them in a timely and efficient manner. This approach is one of continuous discovery, invention, and implementation, with each iteration forcing the development team to drive to closure the project's artifacts in a predictable and repeatable way.

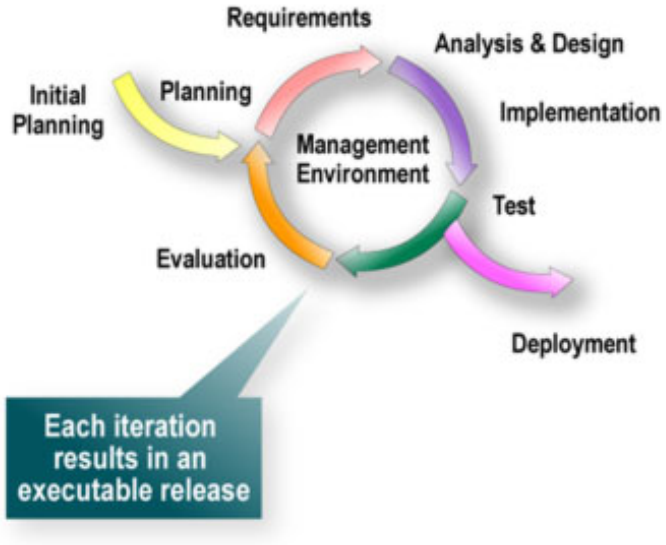


Figure 2: An Iterative Approach to Development

The Good: Benefits of Iterative Development

Compared with the traditional waterfall process, the iterative process has

many advantages.

1. Serious misunderstandings are made evident early in the lifecycle, when it's possible to react to them.
2. It enables and encourages user feedback, so as to elicit the system's real requirements.
3. The development team is forced to focus on those issues that are most critical to the project, and team members are shielded from those issues that distract them from the project's real risks.
4. Continuous, iterative testing enables an objective assessment of the project's status.
5. Inconsistencies among requirements, designs, and implementations are detected early.
6. The workload of the team, especially the testing team, is spread out more evenly throughout the lifecycle.
7. This approach enables the team to leverage lessons learned, and therefore to continuously improve the process.
8. Stakeholders in the project can be given concrete evidence of the project's status throughout the lifecycle.

Risk Mitigation

An iterative process lets you mitigate risks earlier because integration is generally the only time that risks are discovered or addressed. As you roll out the early iterations, you go through all process components, exercising many aspects of the project, including tools, off-the-shelf software, and people skills. Perceived risks will prove not to be risks, and new, unsuspected risks will be discovered.

If a project must fail for some reason, let it fail as soon as possible, before a lot of time, effort, and money are expended. Do not bury your head in the sand too long; instead, confront the risks. Among other risks, such as building the wrong product, there are two categories of risks that an iterative development process helps to mitigate early:

- Integration risks
- Architectural risks

An iterative process results in a more robust architecture because you correct errors over several iterations. Flaws are detected in early iterations as the product moves beyond inception. Performance bottlenecks are discovered at a time when they can still be addressed instead of being discovered on the eve of delivery.

Integration is not one "big bang" at the end of the life cycle; instead, elements are integrated progressively. Actually, the iterative approach that we recommend involves almost continuous integration. What used to be a lengthy time of uncertainty and pain -- taking as much as 40% of the

total effort at the end of a project -- is now broken into six to nine smaller integrations that begin with far fewer elements to integrate.

Accommodating Changes

You can envisage several categories of changes:

- *Changes in requirements*
  An iterative process lets you take into account changing requirements. The truth is that requirements will normally change. Requirements change and "requirements creep" have always been primary sources of project trouble, leading to late delivery, missed schedules, unsatisfied customers, and frustrated developers. But by exposing users (or representatives of users) to an early version of the product, you can ensure a better fit of the product to the task.
- *Tactical changes*
  An iterative process provides management with a way to make tactical changes to the product -- for example, to compete with existing products. You can decide to release a product early with reduced functionality to counter a move by a competitor, or you can adopt another vendor for a given technology. You can also reorganize the contents of an iteration to alleviate an integration problem that needs to be fixed by a supplier.
- *Technological changes*
  To a lesser extent, an iterative approach lets you accommodate technological changes. You can use it during the elaboration phase, but you should avoid this kind of change during construction and transition because it is inherently risky.

Learning as You Go

An advantage of the iterative process is that developers can learn along the way, and the various competencies and specialties are more fully employed during the entire life cycle. For example, testers start testing early, technical writers write early, and so on; in a non-iterative development, the same people would be waiting to begin their work, making plan after plan. Training needs -- or the need for additional (perhaps external) help -- are spotted early during assessment reviews.

The process itself can also be improved and refined along the way. The assessment at the end of an iteration looks at the status of the project from a product/schedule perspective and analyzes what should be changed in the organization and in the process so that performance will be better in the next iteration.

Increased Opportunity for Reuse

An iterative process facilitates reuse of project elements because it is easier to identify common parts as they are partially designed or implemented instead of identifying all commonality in the beginning. Identifying and developing reusable parts is difficult. Design reviews in

early iterations allow architects to identify unsuspected potential reuse and to develop and mature common code in subsequent iterations. It is during the iterations in the elaboration phase that common solutions for common problems are found and patterns and architectural mechanisms that apply across the system are identified.

Better Overall Quality

The product that results from an iterative process will be of better overall quality than are products that result from a conventional sequential process. The system will have been tested several times, improving the quality of testing. The requirements will have been refined and will therefore be more closely related to the users' real needs. And at the time of delivery, the system will have been running longer.

The Hard: Unexpected Downside and Common Traps

Iterative development does not necessarily mean less work and shorter schedules. Its main advantage is to bring more predictability to the outcome and the schedule. It will bring higher quality products, which will satisfy the real needs of end-users, because you will have had time to evolve requirements as well as a design and an implementation.

Iterative development actually involves much more planning and is therefore likely to put more burden on the project manager: An overall plan has to be developed, and detailed plans will in turn be developed for each iteration. It also involves continuous negotiation of tradeoffs between the problem, the solution, and the plan. More architectural planning will also take place earlier. Artifacts (plans, documents, models, and code) will have to be modified, reviewed, and approved repeatedly at each revision. Tactical changes or scope changes will force some continuous replanning. Thus, team structure will have to be modified slightly at each iteration.

Trap: Overly Detailed Planning Up to the End
It is typically wasteful to construct a detailed plan end-to-end, except as an exercise in evaluating the global envelope of schedule and resources. This plan will be obsolete before reaching the end of the first iteration. Before you have an architecture in place and a firm grip on the requirements -- which occurs roughly at the Lifecycle Architecture (LCA) milestone -- you cannot build a realistic plan.

So, incorporate precision in planning commensurate with your knowledge of the activity, the artifact, or the iteration being planned. Near-term plans are more detailed and fine grained. Long-term plans are maintained in coarse-grained format.

Resist the pressure that unknowledgeable or ill-informed management may bring to bear in an attempt to elicit a "comprehensive overall plan." Educate managers, and explain the notion of iterative planning and the wasted effort of trying to predict details far into the future. An analogy that is useful: a car trip from New York to L.A. You plan the overall route but only need detailed driving instructions to get you out of the city and

onto the first leg of the trip. Planning the exact details of driving through Kansas, let alone the arrival in California, is unnecessary, as you may find that the road through Kansas is under repair and you need to find an alternate route, etc.

Acknowledging Rework Up Front

In a waterfall approach, too much rework comes at the very end, as an annoying and often unplanned consequence of finding nasty bugs during final testing and integration. Even worse, you discover that most of the cause of the "breakage" comes from errors in the design, which you attempt to palliate in implementation by building workarounds that lead to more breakage.

In an iterative approach, you simply acknowledge up front that there will be rework, and initially a lot of rework: As you discover problems in the early architectural prototypes, you need to fix them. Also, in order to build executable prototypes, stubs and scaffolding will have to be built, to be replaced later by more mature and robust implementations. In a healthy iterative project, the percentage of scrap or rework should diminish rapidly; the changes should be less widespread as the architecture stabilizes and the hard issues are being resolved.

Trap: Project Not Converging
Iterative development does not mean scrapping everything at each iteration. Scrap and rework has to diminish from iteration to iteration, especially after the architecture is baselined at the LCA milestone. Developers often want to take advantage of iterative development to do gold plating: to introduce yet a better technique, to perform rework, etc. The project manager has to be vigilant so as to not allow rework of elements that are not broken -- that are OK or good enough. Also, as the development team grows in size, and as some people are moved around, newcomers are brought in. They tend to have their own ideas about how things should have been done. Similarly, customers (or their representatives in the project: marketing, product management) may want to abuse the latitude offered by iterative development to accommodate changes, and/or to change or add requirements with no end. This effect is sometimes called "Requirements Creep."Again, the project manager needs to be ruthless in making tradeoffs and in negotiating priorities. Around the LCA milestone, the requirements are baselined, and unless the schedule and budget are renegotiated, any change has a finite cost: Getting something in means pulling something out. And, remember that "Perfect is the enemy of good." (Or in French: "Le mieux est l'ennemi du bien.")

Trap: Let's Get Started; We'll Decide Where to Go Later
Iterative development does not mean perpetually fuzzy development. You should not simply begin designing and coding just to keep the team busy or with the hope that clear goals will suddenly emerge. You still need to define clear goals, put them in writing, and obtain concurrence from all parties; then refine them, expand them, and obtain concurrence yet again. The bright side is that in iterative development, you need not have all the requirements stated before you start designing, coding, integrating, testing, and validating them.

Trap: Falling Victim to Your Own Success
An interesting risk comes near the end of a project, at the moment the "consumer bit" flips. By this we mean that the users go from believing that nothing will ever be delivered to believing that the team might actually pull it off. The good news is that the external perception of the project has shifted: whereas on Monday the users would have been happy if anything were delivered on Tuesday, they become concerned that not everything will be delivered. This is the bad news. Somewhere between the first and second beta, you find yourself inundated with requests for features that people want to be sure are included in the first release. Suddenly, these become major issues. The project manager goes from worrying about delivering minimal acceptable functionality to a situation in which every last requirement is now "essential" to the first delivery. It is almost as though, when this bit flips, all outstanding items get elevated to an "A" priority status. The reality is that there is still the same number of things to do, and the same amount of time in which to do them. While external perceptions may have changed, prioritization is still very, very important.

If, at this crucial moment, the project manager loses his nerve and starts to cave in to all requests, he actually puts the project in schedule danger again! It is at this point that he or she must continue to be ruthless and not succumb to new requests. Even trading off something new for something taken out may increase risk at this point. Without vigilance, one can snatch defeat from the jaws of success.

Putting the Software First

In a waterfall approach, there is a lot of emphasis on "the specs" (i.e., the problem-space description) and getting them right, complete, polished, and signed-off. In the iterative process, the software you develop comes first. The software architecture (i.e., the solution-space description) needs to drive early lifecycle decisions. Customers do not buy specs; it is the software product that is the main focus of attention throughout, with both specs and software evolving in parallel. This focus on "software first" has some impact on the various teams: Testers, for example, may be used to receiving complete, stable specs, with plenty of advance notice to start testing, whereas in an iterative development, they have to begin working at once, with specs and requirements that are still evolving.

Trap: Too Much Focus on Management Artifacts
Some managers say, "I am a project manager, so I should focus on having the best set of management artifacts I can; they are key to everything." Not quite true! Although good management is key, the project manager must ensure in the end that the final product is the best that can be produced. Project management is not an exercise in covering yourself by showing that you have failed despite the best possible management. Similarly, you may focus on developing the best possible spec because you have been hurt by poor requirements management in the past; this will be of no use whatsoever if the corresponding product is buggy, slow, unstable, and brittle.

Hitting Hard Problems Earlier

In a waterfall approach, many of the hard problems, the risky things, and the real unknowns are pushed to the right in the planning process, for resolution during the dreaded system integration activity. This leaves the first half of the project as a relatively comfortable ride, where issues are dealt with on paper, in writing, without involving many stakeholders (testers, etc.), hardware platforms, real users, or the real environment. And then suddenly, the project enters integration Hell, and everything breaks loose. In iterative development, planning is mostly based on risks and unknowns, so things are tough right from the onset. Some hard, critical, and often low-level technical issues have to be dealt with immediately, rather than pushed out to some later time. In short, as someone once said to me: In an iterative development you cannot lie (to yourself or to the world) very long. A software project destined for failure should meet its destiny earlier in an iterative approach.

One analogy is a university course in which the professor spends the first half of the semester on relatively basic concepts, giving the impression that it is an easy class that allows students to receive good marks at the mid-term with minimal effort. Then suddenly, acceleration occurs as the semester comes to a close. The professor tackles all the challenging topics shortly before the final exam. At this point, the most common scenario is that the majority of the class buckles under the pressure, performing lamentably on the final exam. It is amazing that otherwise intelligent professors are taken aback by this repeated disaster, year after year, class after class. A smarter approach would be to front-load the course, tackling 60% of the work prior to the mid-term, including some challenging material. The correlation to managing an iterative project is to not waste precious time in the beginning solving non-problems and accomplishing trivial tasks. The most common reason for technical failure in startups: "They spent all their time doing the easy stuff."

Trap: Putting Your Head in the Sand
It is often tempting to say, "This is a delicate issue, a problem for which we need a lot of time to think. Let us postpone its resolution until later, which will give us more time to think about it." The project then embarks on all the easy tasks, never dedicating much attention to hard problems. When it comes to the point at which a solution is needed, hasty solutions and decisions are taken, or the project derails. You want to do just the opposite: tackle the hard stuff immediately. I sometimes say, "If a project must fail for some reason, let it fail as soon as possible, before we have expended all our time and money."

Trap: Forgetting About New Risks
You performed a risk analysis at the inception and used it for planning, but then forgot about risks that develop later in the project. And they come back to hurt you later. Risks should be re-evaluated constantly, on a monthly, if not weekly, basis. The original list of risks you developed was just tentative. It is only when the team starts doing concrete development (software first) that they will discover many other risks.

Clashes Because of Different Lifecycle Models

The manager of an iterative project will often see clashes between his

environment and other groups such as top management, customers, and contractors, who have not adopted -- or even understood the nature of -- iterative development. They expect completed and frozen artifacts at key milestones; they do not want to review requirements in small installments; they are shocked by rework; and they do not understand the purpose or value of some ugly architectural prototype. They perceive iteration as just fumbling purposelessly, playing around with technology, developing code before specs are firm, and testing throwaway code.

At a minimum, make your intentions and plans clearly visible. If the iterative approach is only in your head and on a few whiteboards shared with your team, you will run into trouble later on.

The project manager must protect the team from external attacks and politics in order to prevent the outside world from disrupting or discouraging the team. He or she must act as a buffer. In order to be "the steady hand on the tiller," the project manager must build trust and credibility with the external community. Therefore, visibility and "tracking to plan" is still important, especially in light of "the plan" being somewhat unconventional in many people's eyes. In fact, it is actually more important.

Trap: Different Groups Operating on Their Own Schedules
It is better and easier to have all groups (or teams, or subcontractors) operating according to the same phase and iteration plan. Often project managers see some benefit in fine-tuning the schedule of each individual team, each of which ends up having its own iteration schedule. When this happens, all the perceived benefits will be lost later, and teams will be forced to synchronize to the slower group. As much as is feasible, put everybody on the same heartbeat.

Trap: Fixed-Price Bidding During Inception
Many projects are pushed into bidding for contractual development far too early, somewhere in the middle of inception. In an iterative development, the best point in time for all parties to do such bidding is at the LCA milestone (end of elaboration). There is no magic recipe here: It takes some negotiation and education of the stakeholders, showing the benefits of an iterative development, and eventually a two-step bidding process.

Accounting for Progress Is Different

The traditional earned-value system to account for progress is different, since artifacts are not complete and frozen, but are reworked in several increments. If an artifact has a certain value in the earned value system, and you get credit for it at the first iteration in which you created it, then your assessment of progress is overly optimistic. If you only get credit when it becomes stable two or three iterations later, your measure of progress becomes depressingly pessimistic. So when using such an approach to monitor progress, artifacts must be decomposed in chunks. For example: initial document (40%), first revision (25%), second revision (20%), final document (15%). Each chunk must be allocated a value. You can then use the earned value system without having to complete each element.

An alternative would be to organize the earned value around the iterations themselves, and gauge the value from the evaluation criteria. Then the intermediate tracking points (usually monthly) reported in the Status Assessment would be built around the Iteration Plan. This requires a finer-grained tracking of artifacts than the traditional requirements spec, design spec, etc., because you are tracking the completion of various use cases, test cases, and so on.

As Walker Royce says, "A project manager should be more focused on measuring and monitoring changes: changes in requirements, in the design, in the code, than in counting pages of text and lines of code." (See References and "Further Reading" below.) And Joe Marasco adds, "Look out not only for change, but also for churn. Things that change multiple times to return to the same starting point are a symptom of deeper problems."

On the positive side, having concrete software that runs early can be used by the wise project manager to obtain some early credibility points. It can show off progress in a more meaningful fashion than completed and reviewed paperwork with hundreds of check boxes ticked off. Also, engineers prefer "demonstrations of how it works" to "documentation of how it should work." Demonstrate first, then document.

Deciding on Number, Duration, and Content of Iterations

What do we do first? The manager who is new to iterative development often has a hard time deciding on the content of iterations. Initially, this planning is driven by risk, technical and programmatic, and by criticality of the functions or features of the system under construction. (RUP gives guidelines for deciding the number and duration of iterations.) The criteria also evolve throughout the lifecycle. In construction, planning is geared to completing certain features or certain subsystems; in transition, it is geared to fixing problems and increasing robustness and performance.

Trap: Pushing Too Much in the First Iteration

We talked above about not tackling the hard problems first. On the other hand, going too far in the opposite direction is also a recipe for failure. There is a tendency to want to address too many issues and cover too much ground in the first or first few iterations. This fails to acknowledge other factors: A team needs to be formed (trained), new techniques need to be learned, and new tools need to be acquired. And often, the problem domain is new to many of the developers. This often leads to a serious overrun of the first iteration, which may discredit the entire iterative approach. Or, the iteration is aborted -- declared done when nothing runs -- which is basically declaring "victory" at a point at which none of the lessons may be drawn, missing most of the benefits of iterative development.

When in doubt, or when confronted with crisis, make it smaller somehow (this applies to the problem, the solution, the team). Remember that completeness is a late lifecycle concern. "Appropriate incompleteness" should be the manager's early lifecycle concern. If the first iteration

contains too many goals, split it into two iterations, and then ruthlessly prioritize which objectives to attempt to achieve first.

It is better to shoot for a simpler, more conservative goal early in the project. Note we didn't say easy. Having a solid, acquired result early in the process will help build morale. Many projects that miss the first milestone never recover. Most that miss it by a lot are doomed despite subsequent heroic efforts. Plan to make sure you don't miss an early milestone by a lot.

Trap: Too Many Iterations
First, a project should not confuse the daily or weekly builds with iterations. Since there is fixed overhead in planning, monitoring, and assessing an iteration, an organization that is unfamiliar with this approach should not attempt to iterate at a furious rate on its first project. The duration of an iteration should also take into consideration the size of the organization, its degree of geographic distribution, and the number of distinct organizations involved. Revisit our "six plus or minus three" rule of thumb.

Trap: Overlapping Iterations
Another very common trap is to make iterations overlap too much. Starting to plan the next iteration somewhere toward the last fifth of the current iteration, while attempting to have a significant overlap of activities (i.e., starting detailed analysis, designing and coding the next iteration before finishing the current one and learning from it) may look attractive when staring at a GANTT chart, but will lead to problems. Some people will not be committed to following up and completing their own contribution to the current iteration; they may not be very responsive to fixing things; or they will just decide to take any and all feedback into consideration only in the next iteration. Some parts of the software will not be ready to support the work that has been pushed forward, etc. Although it is possible to divert some manpower to perform work unrelated to the current iteration, this should be kept minimal and exceptional. This problem is often triggered by the narrow range of skills of some of the organization's members, or a very rigid organization: Joe is an analyst, and this is the only thing he can or wants to do; he does not want to participate in design, implementation, or test. Another negative example: A large command and control project has its iterations so overlapped that they are basically all running in parallel at some point in time, requiring management to split the entire staff across iterations, with no hope of feeding back lessons learned from the earlier ones to the later ones.

See Figure 3 for a few common unproductive iteration patterns.

A Good Project Manager and a Good Architect

To succeed, a software project needs both a good project manager and a good architect. The best possible management and iterative development will not lead to a successful product without a good architecture. Conversely, a fantastic architecture will fail lamentably if the project is not well managed. It is therefore a matter of balance, and focusing solely on

project management will not lead to success. The project manager cannot simply ignore architecture: It takes both architecture expertise and domain expertise to determine the 20% of things that should go into early iterations.

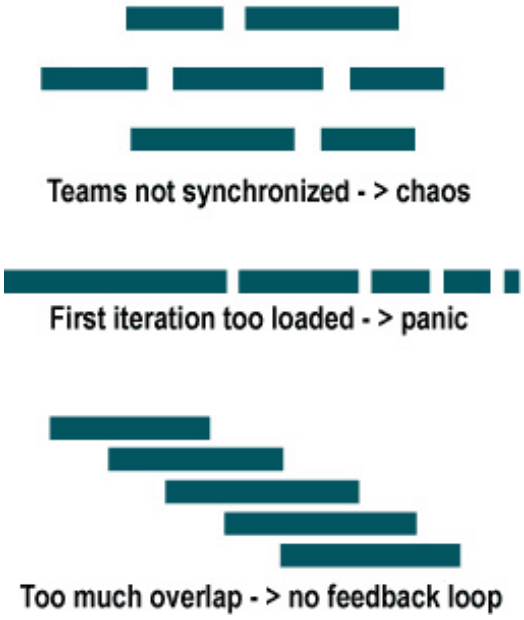


Figure 3: Some Dangerous Iteration Patterns

Trap: Use the Same Person as the PM and the Architect

Using the same person as project manager *and* architect will work only on small projects (5-10 people). For larger endeavors, having the same person play the role of both project manager and architect will usually end with the project neither properly managed nor well architected. First, the roles require different skill sets. Second, the roles, in and of themselves, are more than a full-time job. Therefore, the project manager and architect must coordinate daily, communicate with one another, and compromise. The roles are akin to that of a movie director and a movie producer. Both work toward a common goal but are responsible for totally different aspects of the undertaking. When the same person plays two roles, the project rarely succeeds.

Conclusion

At this stage, you may feel discouraged: so many problems ahead, so many traps to fall into. If it is so hard to plan and execute an iterative development, why bother? Rejoice; there are recipes and techniques to systematically address all these issues, and the payoffs are greater than the inconvenience in terms of achieving reliably higher quality software products. Some key themes: "Attack the risks actively or they will attack you." (From Tom Gilb's book, listed under References and Further Reading.) Software comes first. Acknowledge scrap and rework. Choose a project manager and an architect to work together. Exploit the benefits of iterative development.

The waterfall model made it easy on the manager and difficult for the engineering team. Iterative development is much more aligned with how software engineers work, but at some cost in management complexity. Given that most teams have a 5-to-1 (or higher) ratio of engineers to managers, this is a great tradeoff.

Although iterative development is harder than traditional approaches the first time you do it, there is a real long-term payoff. Once you get the hang of doing it well, you will find that you have become a much more capable manager, and you will find it easier and easier to manage larger, more complex projects. Once you can get an entire team to understand

and think iteratively, the method scales far better than traditional approaches.

Author Note: *John Smith, Dean Leffingwell, Joe Marasco, and Walker Royce helped me write this article by sharing their experiences in iterative project management. Part of this article is included in Chapter 6 of our colleague Gerhard Versteegen's new book on software development (see below).*

References and Further Reading

Rational Unified Process 2000, Rational Software, Cupertino, CA, 2000.

Boehm, Barry W. "A Spiral Model of Software Development and Enhancement," Computer, May 1988, IEEE, pp. 61-72.

Gilb, Tom. *Principles of Software Engineering Management*, Addison-Wesley, 1988.

Kruchten, Philippe. *The Rational Unified Process -- An Introduction*, Addison Wesley Longman, 1999.

Royce, Walker. *Software Project Management -- A Unified Approach*, Addison Wesley Longman, 1999.

Versteegen, Gerhard. *Projektmanagement mit dem Rational Unified Process*, Springer-Verlag, Berlin, 2000.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!

What Is Real-Time Embedded Software?

by Garth Gullekson
Rational Software

Welcome to the "eManagement" column, where The Rational Edge spotlights management issues and solutions. In this inaugural issue, I would like to define real-time embedded software and describe the major challenges associated with its development. In future columns I will describe in detail these aspects of Rational's approach to real-time embedded software solutions.

Real-Time Embedded Systems Defined

There are two obvious characteristics of real-time embedded systems that we should briefly examine. First of all, they are "real-time" systems, which means they are designed to process information "now, and not later." More specifically, real-time systems must respond to stimuli correctly and in a timely manner. For example, in an airplane the delay between pilot input and changes in control surfaces (e.g., rudders) must occur within certain acceptable time intervals or the plane will not be flyable. This contrasts with many business applications (e.g., payroll), for which time delays can be irritating but are rarely fatal!

Second, we are describing "embedded" systems, which means that their computing power is built into, or embedded in, the system. Embedded processors are usually designed for specific applications rather than general purposes. For example, a telephone system comprises many embedded processors for various functions such as handling terminals, controlling voice and data switches, etc.

Real-Time Embedded Systems in Business

Most real-time systems are embedded, and vice versa, so the industry often uses both terms when referring to this computing domain. Real-time embedded systems span a broad set of application types and sizes. Everything from programmable washing machines to vast distributed telecommunication networks can be classified as real-time embedded systems. Consumer awareness of these "hidden" computers increased substantially during the Y2K crisis, when the volumes of information regarding airline safety, telecommunications integrity, and proper functioning within a host of other industries demonstrated how reliant our

world has become on real-time embedded technology.

The Internet has accelerated the development of real-time embedded systems, in particular for Internet infrastructure (e.g., communication switches and routers) and Internet-connected devices (e.g., PDAs). The embedded-systems market is rapidly moving to connect almost all embedded devices to the Internet, including those in vehicles, home appliances, and medical devices.

Development Challenges

The growing use of real-time embedded software offers a particularly strong example of what Rational calls "the software development paradox": While companies need to reduce the time they spend on development, at the same time they need to deliver higher product quality. Because real-time embedded software is used in the world's most critical systems (e.g., telecommunications, avionics, and medical equipment), software quality cannot be compromised. Yet this software is complex and difficult to develop, and embedded software is often difficult to upgrade. Real-time embedded software designs must address many challenges, including timeliness, event-driven stimuli, concurrency, distribution, dynamic structure, and dependability.

Timeliness is the most obvious requirement of the problem domain. Systems have to provide high overall performance and low latency (that is, minimum response time) to individual stimuli. However, timeliness is certainly not the only design challenge. These systems are highly "event-driven." Rather than simply providing functionality from start to finish based on a single command (e.g., printing a document, calculating income tax owed based on financial data), an event-driven system needs to continuously react to various discrete events such as input from users and messages from hardware peripherals. For example, a telephone call progresses from one state to another (e.g., providing dial tone, analyzing what number the user is dialing, ringing, etc.) as events stimulate the system (e.g., input from the user's terminal). Such input usually comes from various uncoordinated independent sources, and can occur in random order. This makes designing the behavior of software components extremely challenging, because at every state in a component's operation it must be able to respond properly to many different types of events. For example, in the middle of dialing a phone number, a user may hang up.

Most systems are highly concurrent, supporting multiple use cases (i.e., capabilities) running in parallel. The stimuli driving the systems are themselves often highly concurrent (e.g., multiple users of a system). This can cause design problems because software components are often involved in multiple use cases. Not only must the components be designed to handle each individual use case properly, but also they often must support the concurrent execution of multiple use cases. This includes managing the interactions among concurrent use cases. For example, a component handling a terminal in a telephone system must be prepared to properly handle the use case of taking a terminal out of service while simultaneously executing the use case of the terminal being involved in a call. In such a situation, the software must ensure that the terminal is not

taken out of service until the call is completed.

Many systems, especially for e-development, are also distributed, including multiple processors in the same box or geographical distribution across a network. For example, a photocopier exploits the power of multiple processors to control naturally concurrent activities such as paper feed, image capture, and output control. Distribution increases both the concurrency of stimuli to each embedded system and the complexity of the coordination required among systems. For example, the failure of a particular processor should not cause the other processors in a distributed system to fail.

The structure of real-time embedded systems is often highly dynamic, either based on configuration or dynamically as the system is running (e.g., objects representing telephone calls). Most systems have to be highly dependable, and some are life critical.

Development Tool Requirements

In addition to the above problems associated with the real-time embedded software domain, managers need to concern themselves with the following general development tool needs:

- Projects need a completely integrated environment that maximizes development team effectiveness. These environments need to span the complete development lifecycle, including requirements and design, code generation and debugging, configuration management, and ultimately deployment of the applications running on real-time operating systems (RTOSs). Out-of-the-box integration among these key tool elements is critical. Any effort spent by a development group on integration only detracts from the group's effort toward building the application.
- Project success requires a well-understood development process, given the complexity of the software to be developed and the risk of not delivering the product on time.
- Both the development process and tools must be scalable, from small local development teams to the increasingly large and geographically distributed teams so prevalent in today's e-development.

The Rational Approach

Rational is extremely well placed to serve the real-time embedded software market. The company's first product in 1981 was designed for real-time embedded development using the Ada programming language, and Rational has continued to serve leading real-time embedded companies such as Ericsson and Motorola. Fueled by the Internet and technology advances, real-time embedded software is becoming increasingly ubiquitous. And it is obvious from the above discussion that real-time embedded development presents numerous unique challenges that can benefit from an optimized solution.

Rational Software has taken an integrated approach to this problem. Our approach includes a graphical user interface for visual design and a development language (the Unified Modeling Language, or UML) that is optimized for the problem domain. As a complement, we also vend an integrated set of tools to meet complete project needs -- from requirements management through development to configuration management. These offerings are anchored by Rational Rose RealTime, cited by IDC as "a major contender as the de facto standard for real-time embedded software development." To accelerate project startup, reduce development risk, and manage complexity, we offer a proven development process and various services.

Until the next issue, if you have any questions, please contact me at garth.gullekson@rational.com.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!

Keeping Geographically Distributed Development Teams in Sync

by **Ralph Capasso**
Product Manager
Rational Software

Global production, outsourcing, and jointly developed software projects have all contributed to a new trend in the management of software development projects. It's what we call *distributed development*, meaning that a project's development resources -- including development tools, hardware, middleware, and the development staff -- may be geographically distributed over an area as vast as the globe itself. The days of single-site software development projects are rapidly dwindling. In today's global economy, collaborative software projects spanning multiple development locations are becoming the norm rather than the exception. As a result, developers working on today's largest software projects can span as many as 20 distinct development sites, sometimes across several continents.

Furthermore, with the technical job market as competitive as it's ever been, companies are leveraging development resources wherever they are located, even if they're half a world away. It isn't uncommon today for even small- to medium-sized software development projects to consist of two or more clusters of developers working across several distributed locations.

Prior Limitations

As little as ten years ago, even the largest software projects possessed neither the tools nor the infrastructure to make geographically distributed development practical. Wide Area Networks (WANs), when they were available, were not nearly as fast and reliable as they are today, and working across these early WANs as if they were local networks was not a reasonable option.

To complicate matters, configuration management technology a decade ago barely addressed large-team complexity issues at a single location, never mind multiple, loosely connected sites. Projects either avoided the multiple-site scenario altogether, or simply established producer/consumer relationships in which one team would "throw code over the wall" to remote recipients at another location. Providing concurrent access to the

same set of development tools and artifacts in distributed development environments was difficult, and achieving parallel development across distinct sites was nothing short of a dream.

Early attempts at achieving true parallel development across multiple sites frequently resulted in failure. Not only were WANs impractical for remote users to access a central site quickly or reliably; in addition, attempts at replication were inefficient and prone to error, and they did not begin to address critical issues involving automation, synchronization, or change conflict resolution. In fact, many of today's commercial configuration management systems still have difficulty solving the problems of parallel development -- even within a single site.

Laying the Groundwork

As time progressed, WAN technologies became more robust, reliable, and affordable. Connectivity barriers were lowered, yet the lack of support for complex parallel development in most configuration management systems only intensified the problem. WAN improvements increased the temptation to develop globally by providing better connectivity, but the available infrastructure tools weren't equipped to take advantage of the bandwidth.

A handful of robust configuration management systems, like Rational ClearCase, did come along to address the problems of complex parallel development. These tools, coupled with advances in WAN technology, made global parallel development possible but hardly practical. Even with more robust WANs in place, working across these networks from multiple remote clients to centrally located servers only uncovered a new set of performance and reliability concerns. The foundation for a practical solution had been laid, but the solution itself hadn't been realized.

Enter Rational ClearCase MultiSite

Rational ClearCase MultiSite is an add-on to Rational ClearCase, the industry-leading software configuration management solution. As its name suggests, ClearCase MultiSite extends the power of ClearCase's robust configuration management solution across multiple, geographically distributed development sites.

Rational ClearCase MultiSite's replication features allow every development site to have its own "replica" of artifacts, enabling users at each site to work against a local server rather than having to reach across a WAN for access to important data. By working against local servers of replicated data, developers experience neither the performance problems nor the reliability side effects that often accompany deployments that depend on WAN access to centrally located servers. Working in a replicated data model also frees up an organization's bandwidth for other cross-site communication tasks rather than placing unnecessary strain upon the wide-area information artery. WANs need only be utilized to transfer synchronization packets from site to site rather than shouldering the load of network traffic created by remotely located clients.

Furthermore, Rational ClearCase MultiSite provides automatic

synchronization capabilities that enable each replica to be updated with the changes, and only the changes, that have occurred at every other remote site since the last synchronization. ClearCase MultiSite sync schedules are configurable and can be tailored to meet the needs of development sites connected by high-speed WANs, lower speed connections, and even sites without any network connectivity at all.

Rational ClearCase MultiSite also introduces the notion of "mastership" in order to mask the complexity involved with conducting parallel development across many distinct sites. Mastership virtually ensures that no conflicts will occur during the synchronization of parallel changes made by developers working on the same artifacts simultaneously at different sites, while at the same time not inhibiting their ability to view, change, and integrate their work.

So whether you have projects that span two development sites or twenty, Rational ClearCase MultiSite can bring your team together by providing replication, automated synchronization, and a level of flexibility that meets today's distributed development needs. Over five years and fifty thousand users later, ClearCase MultiSite is still providing value to both large and small shops that have distributed configuration management needs. Most ClearCase MultiSite customers claim that the product pays for itself within the first few months of deployment. Realizing this high rate of return is critical in order to compete in today's global economy.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!

Bridging the Gap between Black Box and White Box Testing

by **Brian Bryson**
Technology Evangelist
Rational Software

I was fortunate enough to get married last year. I say "fortunate," as I rarely discover software testing professionals burning up the "Most Wanted Bachelor" charts. It was a wonderful event, followed by a wonderful honeymoon in Fiji. Upon our return, my wife and I went to the bank to convert our remaining Fijian dollars, and as we waited in line, a commotion erupted. The security guard locked the door. The bank had just been robbed.

Within minutes, officers and detectives were on the scene. The investigation began. To hear the descriptions given by the staff and customers, you would think no two people saw the same event. Nevertheless, within 30 minutes the detectives were able to collate the divergent accounts of the event into a single, accurate portrayal of what had transpired and gather a precise description of the perpetrator. Within an hour a suspect had been apprehended.

What's the lesson in this tale? That no one single perspective can fully depict a given event or situation. This is not news. In fact, in 1995, Rational's own Philippe Kruchten described this same problem in an IEEE article entitled "Architectural Blueprints: The '4+1' View Model of Software Architecture[1]." In this article Kruchten describes how five concurrent views are necessary to fully describe a software system. The same is true with testing. No single type of test provides enough information to quantify the quality of a system. Instead, multiple types of testing must be undertaken to fully ascertain a software application's quality.

Black box and white box tests represent two broad categories of test types. Neither in isolation can accurately depict the quality of the system. Together, however, they give testers a much clearer perspective on system quality.

I Thought We Were Testing Software, Not Boxes

The "box" in black box and white box testing refers to the system under test; the color refers to the visibility that the tester has into the inner workings of the system. With black box testing, the tester has no visibility into those inner workings. The tester sees only the interfaces exposed by the system. By contrast, white box testing offers the tester full visibility into how the system works. Think of a soda vending machine. A black box test would involve inserting the money into the machine and verifying that a soda drops out and that correct change is given. A white box test might involve opening the back panel on the machine and manually triggering the switch that drops the soda.

Black box testing is sometimes referred to as functional or behavioral testing, and it offers numerous benefits. In the first place, a black box test validates whether or not a given system conforms to its software specification. In implementation, black box tests introduce a series of inputs to a system and compare the outputs to a pre-defined test specification. They test not only individual system components, but also the integration between them. The tests are architecture independent -- they do not concern themselves with how a given output is produced, only with whether that output is the desired and expected output. Finally, as they require no knowledge of the underlying system, one need not be a software engineer to design black box tests.

White box testing is sometimes referred to as structural testing. Because white box tests involve the individual components of a system, they require an implicit knowledge of the system's inner workings. In implementation, white box tests introduce a given set of inputs to a component or individual function of a system and compare the outputs to an expected result. Testing is generally not done through a user interface, but by using the debugging features of the given development environment.

How to Choose a Black Box or White Box Test

Black box testing is generally performed by QA analysts who are concerned about the predictability of the end-user experience. Their job is to make sure that the application meets customer requirements and that the system does what it's designed to do. But black box tests offer no guarantee that every line of code has been tested. Being architecture independent, black box testing cannot determine the efficiency of the code. Finally, black box testing will not find any errors, such as memory leaks, that are not explicitly and instantly exposed by the application. This stands in sharp contrast to white box testing, in which, given an infinite amount of time, all lines of code can be tested and clues as to the code's relative efficiency can be ascertained. Generally, developers whose time is relatively more expensive than that of QA analysts perform white box testing. White box testing proves insufficient, however, for situations in which testing isolated components may not reveal integration errors with other components.

The Tools

Rational Software has been producing tools for automated black box and automated white box testing for several years. Rational's functional regression testing tools capture the results of black box tests in a script format. Once captured, these scripts can be executed against future builds of an application to verify that new functionality hasn't disabled previous functionality. Rational also offers white box testing tools that 1) provide run-time error and memory leak detection; 2) record the exact amount of time the application spends in any given block of code for the purpose of finding inefficient code bottlenecks; and 3) pinpoint areas of the application that have and have not been executed.

For as long as these tools have been available, the black box testing tools have resided almost exclusively on the QA analyst's machine, while the white box tools have been purchased primarily by developers. There are several reasons for this divide.

In the first place, QA analysts traditionally do not have a coding and debugging environment at their disposal. And even if they did, the argument goes, most analysts would not understand the information output by the tools. Rational believes that this is an artificial barrier, and that QA work can be improved via white box testing tools that do not require access to source code or a development environment. Now, Rational uses a patented Object Code Insertion (OCI) technology to instrument an application's executable files. No source code is required. This approach to the problem also enables Rational tools, Purify Quantify and PureCoverage, to perform white box testing on third party code and controls, for which source code may not otherwise be available.

With the introduction of Rational Test Studio in early 1999, white box testing became integrated with black box testing; since then, QA analysts have been able to perform white box tests at the same time as black box tests. With Rational's OCI technology eliminating the need for source code or a development environment, QA analysts now have visibility into the "black" box.

While QA analysts are running their functional black box tests, structural white box tests for memory leaks, code bottlenecks, or measuring code coverage can also occur, that development teams as the intended audience. In essence, the QA group can now undertake a job which previously had to be completed by an engineer. Given the relative average salary levels for these two populations, this is clearly an efficient optimization.

Great Minds Don't Think Alike -- They Think Together

I don't think it's too much of a stretch to claim that the technological advancement of Rational's tool set is like a marriage of the white box and black box testing roles. That's why I began this article with the anecdote about my own recent marriage. We're still different people, my wife and I, but our goals are more focused now, and we're working together toward some common objectives. Similarly, the more testing professionals can

share their detection tools and information base, the more quickly and accurately they can ascertain the overall quality of a software application.

Testers performing traditional black box tests can leverage these same scripts to reap information from their testing that was previously only available to the white box tester on the development team. In so doing, QA analysts have the opportunity to lighten the load on the development team at very little expense. Earlier, I said that "given an infinite amount of time, all lines of code can be tested." Obviously, no software team has the luxury of an infinite amount of time. However, if one reduces the goal of white box testing to finding memory leaks and application crashes throughout the code base, then the time needed for white box testing becomes much more manageable. By putting a common tool set on both the developer's and the QA analyst's desktops, Rational has not only brought black box testing and white box testing a step closer together -- but it has also brought developers and QA analysts closer together. This more unified team may be the greatest benefit of all.

---

[1]You may access Kruchten's white paper here.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!

Keys to Successful Tool Adoption: Techniques and Training

by **Marsha Sheahen**
Director of Education and Training
Rational University

How does a development team successfully adopt new tools and use them to the best advantage? By combining the technical adoption with training in new development techniques. After all, the two go hand in hand. You can't really use the tools effectively without understanding the techniques for which they were designed. Just as important, when *everything* is new, the organization is more likely to recognize the value of investing time and resources in professional training. And there's no doubt about it: A serious investment in tools demands an equally serious investment in training to ensure the greatest benefit from those tools.

Too often, organizations put sophisticated tools in the hands of developers without giving these people any understanding of *why* they are being asked to perform new functions. Trying to use a visual modeling tool, for example, without knowing the modeling notation, object interactions, and the tool's way of depicting these interactions, is a time consuming and challenging proposition for any programmer. And the problem gets compounded when he or she must lead team members, who also do not understand the tool's basic concepts or language, in creating the design for a complex system. Without a common language for interpreting models and designs, the team will have great difficulty meeting the project's component delivery schedule.

It doesn't have to be that way. A well-designed training program can help the whole team learn the methodology, techniques, language, and tools for the project and ensure its success. Additional consulting, project and skills assessments, and services can also be valuable in kickstarting a project that's using new tools.

Of course, it's most important to pick the right project for your introduction. Be realistic. Even with the best training, a team probably can't learn a whole new set of tools and techniques *and* deliver high-quality software in a tighter-than-usual timeframe.

One of our large customers, an organization with about 750 software developers, offers an example of good strategic planning for new tool adoption. They recently selected a suite of new tools and decided to roll them out to their development teams over a 12 month period rather than give them to all the developers at once for use in multiple projects. Initially, they gave the tools to a project team that recently completed a successful release and was planning to develop several new subsystems as product enhancements. The team was eager to move over to object-oriented design and knew they needed development process improvements. Along with training for the new tools, the company provided education in two new techniques: writing use cases to find requirements with Rational RequisitePro, and object-oriented analysis and design with UML notation to provide the foundation for visual modeling with Rational Rose. Then, team members were allowed to adopt these new techniques and tools in a protected environment, while the rest of the organization continued to use traditional development methods.

Later, as other company teams were ready to transition to the new methods, each one received training to get them started. They also had opportunities to learn from the experiences of earlier adopters. This kind of knowledge transfer helps teams become self-sufficient, and therefore more efficient, as they work to bring quality products to market quickly.

For this company, a comprehensive training plan-along with a sound strategy of not biting off more than they could chew -- helped the organization transition smoothly to new, more efficient development methods, and empowered their teams for success. Many companies have also recognized that a commitment to training not only enhances the chances for successful tool adoption, but also demonstrates the company's commitment to employee development and ultimately helps to reduce attrition.

At Rational University, we urge our customers to think about these strategic benefits when they consider a training investment. A software development team armed with a good education and great development tools will clearly have a greater chance of success than a team with the same tools and no education. In addition, a company that recognizes the value of training and invests in its employees will see higher productivity, better products, and lower turnover among their most valuable employees.

About Rational University

Rational University, a business unit of Rational Software Corporation, creates educational materials with the big picture in mind, showing people how and why to use new techniques before teaching them how to use their new tools. In addition to developing courseware that maps to Rational's tools, we address tool integrations and keep pace with the latest thinking from Rational on software development best practices. Rational's top thought leaders--Ivar Jacobson, Grady Booch, Jim Rumbaugh, Philippe

Kruchten and others were charter members of Rational University. Their continued involvement ensures continuity and consistency between the thinking embodied in the Rational Unified Process and our curricula.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!

Managing an Integrated RUP/SDE Implementation

by Gayl Lepore
European Director
Strategic Services Organization
Rational Software

This article introduces a process for managing a project that involves the implementation of the Rational Unified Process (RUP) in conjunction with a software development environment (SDE). The process should be of particular interest to customers who are considering a full implementation of RUP and are looking for additional implementation guidance to complement the Environment Workflow in RUP.

This approach offers the customer a more comprehensive strategy for planning the numerous organizational and technical aspects associated with such a technology implementation effort. For this strategy to succeed, however, the consulting team must ensure that the customer and Rational agree on a joint implementation plan, and that the plan is adaptable to the evolving needs of the project as it progresses through the development lifecycle.

Overall Implementation Strategy

Implementing a software development process change is a substantial undertaking that requires planning and organizational commitment to be successful. A key to success at the organizational level is to break the implementation effort down into three main parts for planning and execution.

1. **Managing the SDE Implementation** -- with a focus on planning and control of the overall SDE implementation effort
2. **Developing an SDE Solution** -- with a focus on creating a RUP adaptation and appropriately configured tooling environment specific to the needs of the project
3. **Deploying the Resulting SDE** -- with a focus on the rollout of the process and tools to project members

Separating the overall undertaking into distinct parts is an effective way to

focus organizational responsibility and apply structure to the myriad technical activities associated with implementing an integrated RUP/SDE.

1. **Managing the SDE Implementation**

   It goes without saying that adopting new technology requires commitment to goals and investment of resources to achieve results. Before you invest resources in the tangible development and deployment of the SDE, you must first define the full context for the integrated RUP/SDE implementation. By defining the overall purpose, goals, scope, and appropriate means for implementation, you clearly define a scope within which to address detailed planning and execution concerns.

   The first step in managing the overall solution, therefore, should focus on addressing the following planning concerns:

   - Vision
   - Strategy
   - Steering
   - Means of execution

   The *vision* is used as a project blueprint for explaining and reiterating, as necessary, the organizational priorities, technical goals, cost/benefit rationale, and implementation success criteria. The *strategy* defines how the vision will be realized appropriately within the scope and timeframe of the project. *Steering* addresses the participants and functions needed to monitor and control the SDE implementation. And *means of execution* covers the allocation of resources and day-to-day coordination -- such as an SDE project or Software Productivity Improvement effort -- that will be used to plan, manage, and execute the RUP/SDE initiative.

2. **Developing an SDE Solution**

   Regardless of how well suited to your requirements a given technology may be "out of the box," the needs of a given project will always require configuration and integration. The scope of the activities involved can be quite substantial when you consider that the integration of RUP and a tool environment will affect an entire software development lifecycle.

   On the process side, the key factors to address in adapting RUP to the project situation are:

   - Defining which of the RUP artifacts the project will use and the content of each artifact. In addition, if the project is evolving an existing system or application that was developed with a different process, the form and content of existing information will need to be related, and possibly

migrated, to the form and content defined by the RUP artifacts.

- Defining which of the RUP roles will be implemented by the project and how to staff each role.
- Defining which activities of the RUP workflows will be performed and what modifications are required to accommodate project organizational needs and technical considerations of the software being developed.

On the tooling side, the key factors to consider in configuring the tools for the project are:

- Implementing a schema for each tool that best supports the artifacts and activities defined by the process.
- Implementing physical interconnections among tools to support the process activities that require information consistency across artifacts, and to support the controlled exchange of artifacts among project members.
- Determining what's required to provision and administer a hardware and networking environment with sufficient performance to run the tools and exchange artifacts among users.
- Determining administrative needs for project members who will use the tools and how these people will be supported when they encounter problems.

For the project as a whole, the principal factors to address are:

- Ensuring that the process and tooling solution has integrity across the whole lifecycle;
- Ensuring that appointed users are involved throughout the process of developing the project solution so that they develop sufficient skills to become mentors during the project rollout.

The end goal of this stage is to create a ready-to-deploy SDE.

3. **Deploying the Resulting SDE**

Having developed a solution, the perspective shifts toward deployment. This part of the implementation effort involves the rollout of RUP and the integrated tool environment to project users. An important goal at this stage is also to plan the eventual self-sufficiency of the target organizations regarding usage, maintenance, and evolution of the SDE.

The critical factors to address when rolling out the environment to the project organization are:

- Physical installation of the SDE for project members.
- Both generally applicable and role-specific training and mentoring for project members.
- Support and maintenance of the process and tooling environment during the project's lifetime.
- Organizational ownership of the process and longer-term evolution of the methods and tools.

The end goals of deployment are capable project users and administrators, and a software organization that is no longer dependent on external consulting for its SDE.

Steps to Successful Project Implementation

Fleshing out each of the above parts associated with SDE implementation represents a first big step in planning and organizing an integrated RUP/SDE initiative. Too often, organizations think that one SDE fits all projects and assume that they can use a master configuration for their project without any additional implementation effort. Or, when they *do* recognize that an implementation effort is required, they plan and attempt to configure the process and tool environment without sufficient understanding of project needs or input from project members. Unfortunately, this leads to SDE results that are untimely or insufficient to meet the software project's needs.

Both of these are common mistakes in full RUP implementations and mostly relate to a key point alluded to in the introduction: the need to use an implementation approach that is flexible enough to adapt to the needs of the project as it progresses. To overcome these pitfalls, the suggestions below provide further guidance on how to successfully implement an integrated SDE for the project.

Achieving an Effective Implementation

As the project progresses, the team responsible for the SDE must be able to incrementally configure and deploy functional pieces of the environment into the project. For example, it should be possible to deploy the process and tooling environment to manage requirements if need be, prior to deploying the environment required to test the system.

Activities to be performed for the SDE should be planned against a well-defined set of intermediate and final implementation deliverables agreed to up front by the overall project. This allows the project leaders to tell the implementation team when certain process and tooling capabilities will be required, to plan the lead-time required for training, and to budget sufficient resources for mentoring. The deliverables themselves should be standardized for each of the functional parts of the environment that the project intends to use. This helps significantly in ensuring consistency between how the process and tools are configured and the performance of the integrated environment once it's deployed.

Finally, the rollout of the process and tools to the project should be carried out by deployment teams who take responsibility to ensure that all members of the project team get the necessary support.

Incremental Configuration Using Standard Deliverables

Planning and execution for incremental SDE configuration is achieved by creating a focused effort for each of the RUP workflows to be implemented. Each such effort is responsible for the development of an integrated process and tool environment to support a given workflow. Each workflow development effort, in turn, is responsible for four deliverables to the project:

- A proof of concept demonstrating the tailored workflow and associated tooling solution proposed for the project.
- A "reference installation" of the process workflow and associated tooling required at each deployment site. A reference installation is a working prototype of the SDE deployed to at least one client in all participating sites, with servers configured as if they were in live use; typically, these servers are connected through the actual network and reside on the target machines they intend to use as hosts. The reference installation is used to ensure that the environment configuration functions as intended in its target environment.
- A training workshop based on the reference deployment so the deployment teams can educate end-users on how to use the tailored process and integrated tool environment for their project.
- An environment specification covering the technical details of the process tailoring and tooling configuration. This is used to facilitate administration and maintenance of the SDE.

Rolling Out with Deployment Teams

A final and critical step to effective implementation of an SDE is the creation of deployment teams who have responsibility for each of the following rollout concerns:

- IS/IT administration and first line support: Deliverables include a technology installation and administration plan, as well as documented support procedures and trouble report handling capability.
- Training and mentoring: Deliverables include definition of a role-based training curriculum and schedule for target users and organizations, as well as the establishment of a mentoring network that supports learning while on the job.
- SDE transition and maintenance: Deliverables include a transition specification and plan so that responsibility to maintain or evolve the SDE after project completion is transitioned to the appropriate organization.

By assigning one or more deployment teams to the rollout, the project ensures that all project members get adequate and consistent support in their use of the process and tools. How many deployment teams a project needs to support a rollout is influenced by the following factors:

- The number of users in the project who will use the SDE. A deployment team of one process engineer and one tooling engineer should be able to support ten active new users in a single location. As users in the project become more experienced, the deployment team can support up to 15 or 20 users in a single location.
- The number of geographic locations associated with the project. Deployment teams are needed in each location, as is an IS/IT support team for the administration of the physical environment.
- A team with responsibility for the maintenance of the RUP/SDE for the project as a whole is usually also required 1) to make fixes and enhancements as the project progresses, and 2) to transition the process and tools to the software organization if future projects intend to re-use the environment.

Summary

Providing a RUP-based software development environment for a project requires a thought-through strategy and comprehensive implementation approach. The approach introduced here has been successfully used for this purpose and is well suited to managing incremental implementation of RUP in the context of an ongoing project.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!

Why Use Cases Are Not "Functions"

by **Kurt Bittner**
General Manager
Rational Unified Process Business Unit

Most people go astray right from the start with use cases. Perhaps it is the similarity between use case diagrams and dataflow diagrams which leads people to define use cases that are simply functions or menu items. Whatever the reason may be, it is notably the most prevalent mistake that novices make.

What's wrong with this picture? In simplest terms, I like to regard a use case as *a story about some way of using a system to do something useful.* Using this definition, are all of these "use cases" independently useful?

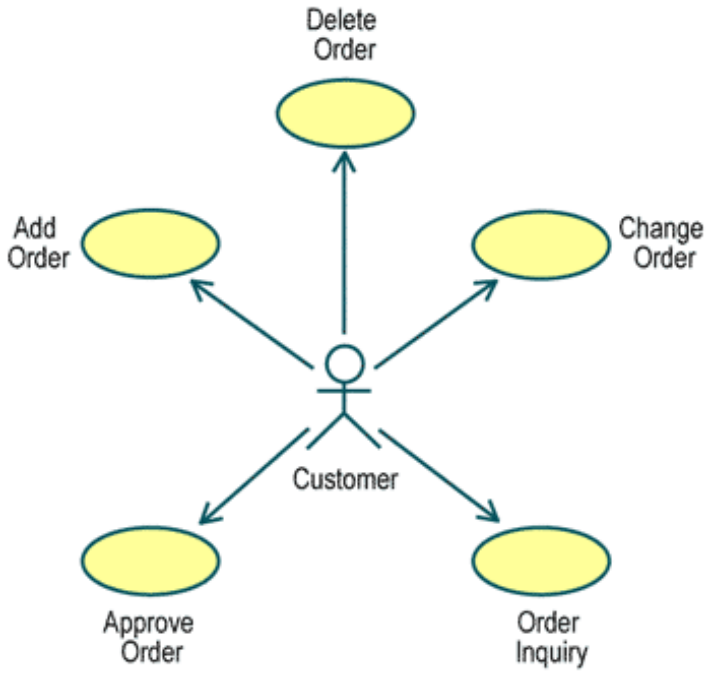


Figure 1: The wrong way: use cases as menu options or functions

The answer, of course, is no. In this example, the use case denotes all things that the system needs to do, but it also represents the one single thing that the customer wants to do on the system: place an order. All of the remaining elements are alternate flows in this one use case. They are steps that may be taken when placing an order. Where there is only one useful thing being done, there should only be one use case. Figure 1 is an example of functional decomposition, or (as one colleague puts it) an example of the "circled wagons" formation -- one actor at the center of a

circle of use cases.

This problem is a common one. Why do people fall into this trap? We have an intrinsic need for order, and where none exists we will impose it if necessary. In the case of functional decomposition, we have a natural tendency to try to break the problem down into smaller and smaller chunks. There is a naive belief that by breaking the use cases into smaller and smaller units, we have simplified the problem. This perception is dead wrong; when we decompose the use cases, we actually compound the problem.

Here's Why

The purpose of a use case is to describe how someone or some thing will use the system to do something that is useful to them. It describes what the system does at a conceptual level so that we can understand enough about the system to decide if the system will do the right thing or not. It enables us to form a conceptual model of the system.

Again, refer back to Figure 1. Now ask yourself, would I want to use this system to inquire into the status of an order if I had never placed an order? It's not very likely. Or would I need to change an order if I had never placed an order? No, probably not. Individually, these things are useful to me only if I have placed an order; all of them are necessary, however, to the system's ability to allow me to place an order.

Decomposing the system into smaller use cases actually *obscures* the real purpose of the system; at the extreme, we end up with lots of isolated odd bits of behavior. As a result, we can't tell *what* the system does. It's just like looking at a car that's been taken apart -- maybe you can tell that it's a car, and you know that the parts must be useful somehow, but you really can't tell how they fit together.

When working with use cases, remember that use cases are a way to think of the overall system and organize it into manageable chunks of functionality -- chunks that do something useful. To get the right set of use cases, ask yourself this question: "What are the actors really trying to do with this system?"

In case you're wondering what the improved version of Figure 1 would look like, the figure below presents the improved version:

Figure 2: A better, simpler approach: combine functions to reflect the real value to the actor

This one use case encompasses all the "functions" that the earlier diagram split out as use cases. You may ask why this is better. The answer is simple. It focuses on the value that the customer wants from the system, not on how we subdivide and structure the functionality

within the system. If you split all these functions into separate use cases, you force *your* customer (the one paying for the system) to reassemble the decomposed use cases into something meaningful to them in order to understand whether the system described is what they want (and are willing to pay for).

Focus on Value

Lots of small use cases are a common problem, especially among teams with a strong background in (or covert sympathies for) functional decomposition. Their use case names read like a list of functions that the system will perform: "Enter Order," "Review Order," "Cancel Order," "Fulfill Order." These may not sound so bad at first, but there are more. For even a small order entry system, use case lists can run well into the hundreds. If one stays on this path, they are soon drowning in a sea of use cases, especially if it is a "really big" system. In this case, you would end up with many hundreds, maybe thousands, of use cases.

So What's So Wrong With This?

The value of these use cases would be lost. *A use case's sole purpose is to result in some sort of value to the actor*, and at one level being able to enter an order is something of value. But if the order could never be fulfilled, would it still have value? Probably not.

Or what about entering an order and modifying the order, or perhaps canceling the order -- all of these things are related to the real thing a customer wants to do, which is to receive the goods being ordered. These actions are also all necessary to what the company wants, which is to receive payment for the goods shipped.

Another problem with a set of functions that appear to be disconnected, without any apparent relationship, is that they result in a hard-to-use system. Too many systems are like this -- they are just jumbles of features. Remember, use cases help us focus on what is really important -- the things that have real value -- and enable us to define a system around those elements. Use cases do *not* present a functionally decomposed picture of the system.

Example

Consider an e-commerce system that you have used on the Web. When you go to the site, your goal may be to find information about products, select products to buy, and arrange payment and shipping terms for those products. In the course of doing those things, you may change your mind, enter incorrect information and have to change it, change your mailing or shipping address, and a number of other things. If the site does not allow you to find products and order them in an appealing way, you probably won't even complete your order, let alone return to the site again.

When building systems, always refer back to the core definition of a use case: a story about some way of using the system to do something useful. If you can implement this definition to display the value that users expect to obtain from the system, and then create use cases that reflect these values, your system will better meet user expectations.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!

Features, Use Cases, Requirements, Oh My!

by **Dean Leffingwell**
Senior Vice President of Process and Project Management
Rational Software

As a follower and proponent of object-oriented (OO) technology in the BU (Before UML) days, I must admit to a certain fascination with the various methods and notations spread by the industry thought leaders at the time. At about two to four years BU, if we had walked into a room full of OO advocates and asked the following question:

> *I think this OO technology shows great promise; but tell me, since the object shares behavior and data, what do you call this thing an object does to fulfill its behavioral obligations?*

We might have gotten the following answers:

> *"It's a responsibility!"* (Wirfs-Brock)
>
> *"It's an operation!"* (Booch)
>
> *"It's a service!"* (Coad/Yourdon)
>
> *"It's a (virtual) function!"* (Stroustrup)
>
> *"It's a method!"* (many others)

And if this array of answers seems confusing, don't even think about the range of responses we would have elicited by asking how you would graphically represent that thing we call an object and a class (e.g., "It's a rectangle," "It's a cloud," "It's a. . . whatever."). While these differences might seem inconsequential, the reality is that some of the most significant shared concepts among our software engineering leaders -- inheritance, relationships, encapsulation -- were obscured by minor differences in terminology and notation. In other words, neither the science of OO engineering nor the benefits to be gained could advance further because the language to describe the science had not yet been invented. Of course, gaining agreement among these authors, methodologists[1], and independent thinkers was not a trivial matter, but eventually, along came the UML, and the science of software

engineering marched forward again.

While it's perhaps not as bad as the Tower of Babel wrought by the pre-UML competing OO methodologies, the methodology of *requirements management* suffers from some of the same issues -- specifically, the prevalence of ambiguous, inconsistent, and overloaded usage of common terms. These terms, including such seminal constructs as "Use Cases," "Features," and "Requirements," we assume are common, everyday terms that "everyone understands." In truth, however, each individual attaches his or her own meaning to these terms within a given context. The result is often ineffective communication. And this occurs in a domain wherein success is defined simply by having *achieved* a common understanding.

Booch [Booch 1994] points out that Stepp observed:

> *. . .an omnipresent problem in science is to construct meaningful classifications of observed objects and situations. Such classifications facilitate human comprehension of the observations and subsequent development of a scientific theory.*

In order to advance the "scientific theory" of requirements, we have to come to terms with terms!

The purpose of this article is to take a small step forward in the discipline of software engineering by defining and describing some of the most common terms and concepts used in describing requirements for systems that contain software. In doing so, we hope to provide a basis for common understanding among the many stakeholders involved: users, managers, developers, and others. Certainly if we communicate more effectively and establish a common view, it will be possible to more quickly develop and deliver higher quality systems.

This article is not an overview of the requirements management discipline -- for that we refer you to a number of books on the topic listed under the heading "Suggested Reading." The goal of this article is simply to help practitioners in the field improve their ability to answer the following, fundamental question:

> *"What, exactly, is this system supposed to do?"*

The Problem Domain vs. the Solution Domain

Before we start describing specific terms, however, it's important to recognize that we will need to define terms from two quite different worlds -- the world of the problem and the world of the solution. We'll call these the *problem domain* and *solution domain*, respectively.

The Problem Domain

If we were to fly over the **problem domain** at a fairly low level, we would see things that look very much like the world around us. If we

flew over the HR department, we might see employees, payroll clerks, and paychecks. If we flew over a heavy equipment fabricator, we might see welders, welding controllers, welding robots, and electrodes. If we flew over the World Wide Web, we'd see routers and server farms, and users with browsers, telephones, and modems. In other words, in any particular problem domain we can most readily identify the things (entities) that we can see and touch. Occasionally, we can even see relationships among those things; for example, there seems to be a one-to-one relationship between Web users and browsers. We might even see messages being passed from thing to thing -- e.g., "That welder appears to be programming a sequence into a welding robot's 'brain.'"

If we were really observant, we might see things that look like *problems* just waiting to be resolved: "The welder seems really frustrated with his inability to get the sequence right," or "Notice that nasty time delay between the time that employee enters her payroll data and the day she receives her check!"

Some of the problems seem to just *beg* for a solution. So we say: "Perhaps we can build a system (a better programmable controller, a more efficient payroll processing) to help those poor users down there fix those problems."

On User and Stakeholder Needs

Before we build that new system, however, we need to make sure that we understand the *real needs* of the users in that problem domain. If we don't, then we may discover that the welder was grimacing only because he was suffering from a painful corn on his toe, so neither he nor his manager is interested in purchasing our brand new "SmartBot" automated welding control unit. We might also notice that when we try to sell the SmartBot, the manager seems to emerge as a key stakeholder in the purchasing decision. We don't remember seeing her in our fly-over. (Perhaps she was in the smoking lounge; our cameras don't work as well in there.) In other words, not all stakeholders are users, and we have to understand the needs of both communities (stakeholders and users) if we hope to have a chance to sell the SmartBot. To keep things simple, we call all of these needs *stakeholder needs*, but we'll constantly remind ourselves that the potential users of the system appear to represent a very important class of stakeholders indeed.

We'll define a stakeholder need as:

> *. . .a reflection of the business, personal, or operational problem (or opportunity) that must be addressed to justify consideration, purchase, or use of a new system.*

Stakeholder needs, then, are an expression of the issues associated with the problem domain. They don't define a solution, but they provide our first perspective on what any viable solution would need to accomplish. For example, if we interview the plant manager for a heavy

equipment fabricator, we may discover that welding large, repetitive weldments consumes a significant amount of manufacturing time and cost. In addition, welders don't seem to like these particular jobs, and they are constantly in danger of burnout. Worse still, the physical aspects of the job -- repetition, awkward manual positions, danger to eyesight, and so on -- present personal safety issues and long-term healthcare concerns.

With these insights, we could start defining some *stakeholder needs:*

> *We need an automated way to fabricate large, repetitive weldments without the welder having to manually control the electrode.*
>
> *We are happy to have a welder present, but we need to remove him to a safety zone outside of the welding area and away from any moving machinery.*
>
> *We need an easy-to-use "training mode" so that average welders can "train" the machine to do the majority of the welding for them.*
>
> *We need to allow more flexibility in the training mode and recognize that this may contradict some aspects of the need for user-friendliness.*

As we understand these various aspects of the system, we'll mentally "stack" these discoveries in a little pile called "stakeholder needs."

The Solution Domain

Fortunately, our fly-over of the problem domain doesn't take very long, and (usually) what we find there is not too complicated. We start to appreciate the problem when we leave the airplane, and set off to build a solution to the problems and needs we have observed. Yes, we've reached the beginning of the hard part: *forming a solution to the problem*. We consider the set of activities (system definition, design, and so on), the "things" we find and build to solve the problem (computers, robot arms, and the like), and the artifacts we create in the process (such as source code, use cases, and tests) part of the **solution domain**.

In the solution domain, there are many steps and activities we must successfully execute to define, build, and eventually deploy a successful solution to the problem. They include:

1. Understand the user's needs
2. Define the system
3. Manage the scope and manage change
4. Refine the system definition
5. Build the right system

In a nutshell, the steps above define a simplified process for requirements management. This paper won't discuss these steps in much detail; for this we refer you to the Bibliography and Suggested Reading, including the text, *Managing Software Requirements* [Leffingwell, 1999]. The ideas in this paper are consistent with those in that text, and most of the definitions provided here are taken from it.

The text defines requirements management as:

> *. . .a systematic approach to eliciting, organizing, and documenting the requirements of the system, and a process that establishes and maintains agreement between the customer and the project team on the changing requirements of the system.*

But let's move on to discovering and defining more of the requirements management terms we'll need to describe the system we are about to build.

Common Requirements Terms in the Solution Domain

Features of a Product or System

As we start thinking about solutions to the problems we've identified, it's very natural to start jotting down the *features* of a system. Features occupy an interesting place in the development of a system. They fit somewhere between an expression of the user's real needs and a detailed description of exactly how the system fulfills those needs. As such, they provide a handy construct -- a "shorthand", if you will -- for describing the system in an abstract way. Since there are many possible solutions for the problem that needs to be solved, in a sense features provide the initial bounds of a particular system solution; they describe what the system is going to do and, by omission, what it will not do.

We'll define a feature as:

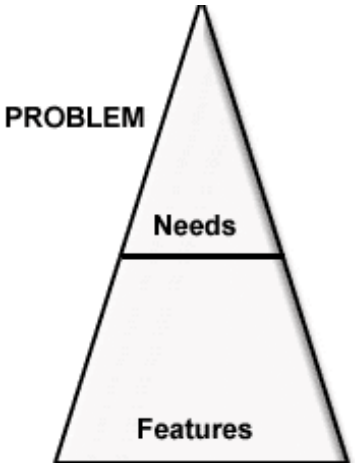
> *. . .a service that the system provides to fulfill one or more stakeholder needs.*

Features are easily represented in natural language, using terms familiar to the user. For example:

> *The system runs off standard North American power.*

> *The tree browser provides a means to organize the defect information.*

> *The home lighting control system has interfaces to standard home automation systems.*



Since features are derived from stakeholder needs, we position them at the next layer of the pyramid, below needs. Note that we've also moved from the problem domain (needs) to the first level of the solution domain (features).

It's important to notice that features are NOT just a refinement (with increasing detail) of the stakeholder needs. Instead, they are a direct response to the *problem* offered by the user, and they provide us with a top-level *solution* to the problem.

Typically, we should be able to describe a system by defining 25-50 features that characterize the behavior of that system. If we find ourselves with more than 50 features on our hands, it's likely that we've insufficiently abstracted the true features of the system. Or the system may be too large to understand, and we may need to consider dividing it into smaller pieces.

Features are described in natural language so that any stakeholder who reads the list can immediately gain a basic understanding of what the system is going to do. A features list usually lacks fine-grained detail. That's all right. Its purpose is simply to communicate the intent and, since many stakeholders are likely to be non-technical, too much detail can be confusing and may even interfere with understanding. For example, a partial list of features for our SmartBot automated welding robot might include:

> *A "lead through path" training mode that allows the welder to teach the robot what paths will be welded.*
>
> *A "step-and-repeat" feature that supports repetitive welding sequences.*

Use Cases

As we think further about the way in which the system needs to do its job for the user, we might find it beneficial to employ the *Use Case Technique* for further describing system behavior. This technique has been well developed in a number of books [Jacobson 1992] and is also an integral technique in the industry-standard Unified Modeling Language (UML) [Booch 1999].

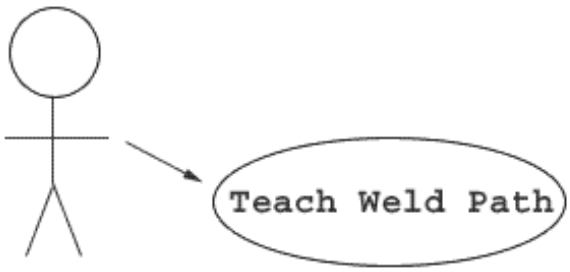
Technically, a use case:

> *. . .describes a sequence of actions, performed by a system, that yields a result of value to the user.*

In other words, the use case describes a series of user and system interactions that help users accomplish something they wanted to accomplish. Stated differently, the use case describes HOW users and

the system work together to realize the identified feature.

Use cases also introduce the construct of an actor, which is simply a label for someone who is using the system at a given time. In UML, a use case is represented by a simple oval; an actor is represented by a stick figure with a name. So we can illustrate both with a simple diagram like the one below.



The use case technique prescribes a simple, step-by-step procedure for how the actor accomplishes the use case. For example, a use case for Step and Repeat might start out as follows:

> Step 1: The welder presses the "Step and Repeat" button to initiate the sequence.
>
> Step 2: The welding system releases power to the drive motors so that the robot's arms can be moved manually.
>
> Step 3: The welder grabs the trigger, moves the arm to the weldment, and holds down the "Weld Here" button for each path to be welded.

The use case technique provides a number of other useful constructs, such as pre and post descriptions, alternate flows, and so on. We'll talk about these later as we examine the use case in more detail. For now, we simply need to know that use cases provide an excellent way to describe *how* the features of the system are achieved.

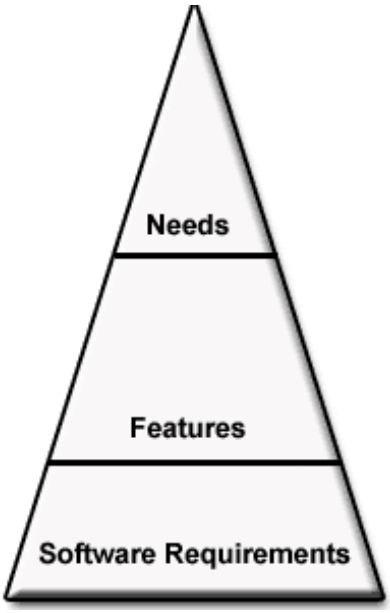
For planning purposes, it's likely that more than use cases will be necessary to describe how a particular feature is implemented. A small number of use cases (perhaps 3-10) may well be necessary for each feature. In describing the use cases, we are elaborating on the behavior of the system. Detail increases as we achieve additional specificity.

Vision Document

Many development projects use a Vision document that defines the problem, identifies key stakeholders and user needs, lists system features, and perhaps includes example use cases. This document may be called by a variety of other names: Project Charter, Product Requirements Document, Marketing Requirements Document, and so forth. No matter what it's called, the Vision document highlights the overall intent and purpose of the system being built. It captures the "gestalt" of the system, using *stakeholder needs*, *features*, and *use cases* to communicate the intent.

We cannot, however, simply dump features and initial use cases into the hands of the development team and expect them to rush off and develop a system that really satisfies stakeholder needs. We need to be a lot more definitive about what we want the system to do, and we'll probably have to add a lot of new stakeholders, including developers, testers, and the like. That's what happens in the next layer of the system definition -- the software requirements.

Software Requirements



Software requirements provide the next level of specificity in the requirements definition process. At this level, we specify requirements and use cases sufficiently for developers to write code and testers to see whether the code meets the requirements. In our graphical representation, software requirements are at the base of our pyramid.

What is a software requirement? Although many definitions have been used throughout the years, we find the definition provided by requirements engineering authors Dorfmann and Thayer [Dorfmann 1990] to be quite workable. They say that a software requirement is:

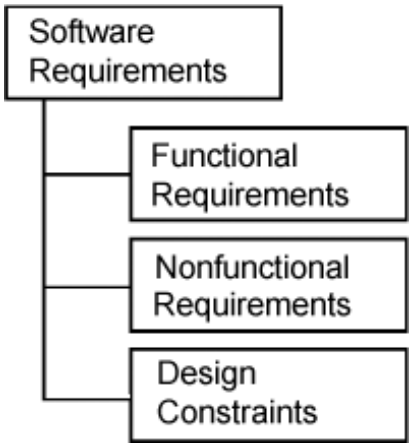
> *. . .a software capability needed by the user to solve a problem that will achieve an objective*
>
> *OR*
>
> *a software capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification or other formally imposed documentation.*

Applying this definition, the team can develop a more specific set of requirements to refine, or elaborate, the features list discussed earlier. Each requirement serves some feature and vice versa. Notice the simplicity of this approach. We have a list of features, and we then elaborate those features by writing a set of requirements that serve those features. *We don't write any other requirements.* We avoid the

temptation to sit down, stare at the ceiling, and "think up some requirements for this system."

The process is straightforward but not necessarily easy. Each feature is reviewed, and then requirements are written to support it. Inevitably, writing the requirements for one feature will spur ideas for new requirements or revised requirements for a feature that has already been examined.

Of course, as we know, it's not easy to write down requirements -- and there may be a large number of them. It's helpful to think about three types or categories of software requirements: functional requirements, nonfunctional requirements, and design constraints.



We find these three categories helpful in thinking about the requirement and what role we expect it to fill. Let's look at these different types of requirements and see how we can use them to define different aspects of the proposed system.

Functional Requirements

Functional requirements express what the system does. More specifically, they describe the inputs and outputs, and how it is that specific inputs are converted to specific outputs at various times. Most business software applications are rich with functional requirements. When specifying these requirements, it's important to strike a balance between being too vague ("When you push the 'On' button, the system turns on") and being too specific about the functionality. It's important to give designers and implementers as wide a range of design and implementation choices as possible. If we're too wishy-washy, the team won't know what the system is supposed to achieve; if we're too specific, we may impose too many constraints on them.

There isn't one right way to specify requirements. One technique is simply to take a declarative approach and write down each detailed thing the system needs to do. For example:

> During the time in which the "Weld Here" input is active, the system digitizes the position of the electrode tip by reading the optical encoders every 100 msec.

Elaborating the Use Case

In many systems, it's helpful to organize the specification activity by refining the use cases defined earlier and developing additional use cases to fully elaborate the system. Using this technique, we refine the steps of the use case into more and more detailed system interactions. We'll also need to define pre-conditions and post-conditions (states the system assumes before and after the use case), alternative actions due to exception conditions, and so on.

Since use cases are semantically well defined, they provide a structure

into which we can organize and capture the system behavior. Here is a representative use case for the Smartbot.

Use Case Name	**Teach Weld Path**
Actor	Welder
Brief Description	This use case prescribes the way in which the welder teaches the robot a single weldment path operation.
Flow of Events	Basic flow for the use case begins when the welder presses the "Teach" button on the control console. The system turns off the power to the robot arms. The welder grabs the teaching electrode and positions the teaching tip at the start of the first weld. The welder presses the "Weld Here" trigger and simultaneously moves the teaching tip across the exact path to be welded. At the end of the path, the welder releases the "Weld Here" trigger and then returns the robot's arm to the rest position.
Alternative Flow of Events	At any time during the motion, the welder can press the "Pause" button; then the robot will turn on power to the motors and hold the arms and teaching tip in the last known position.
Pre-conditions	The robot must have performed a successful auto-calibrate procedure.
Post-conditions	The traverse path and weld paths are remembered by the system.
Special Requirements	The welder cannot move the tip at a rate faster than 10cm/second. If faster motion is detected, the system will add resistance to the arms until the welder returns to the acceptable lead-through speed.

Nonfunctional Requirements

In addition to functional requirements such as inputs translating to

outputs, most systems also require the definition of a set of nonfunctional requirements that focus on specifying additional system "attributes," such as performance requirements, throughput, usability, reliability, and supportability. These requirements are just as important as the input-output oriented functional requirements. Typically, nonfunctional requirements are stated declaratively, using expressions such as "The system should have a mean time between failure of 2,000 hours;" "The system shall have a mean time to repair of 0.5 hours;" and "The Smartbot shall be able to store and retrieve a maximum of 100 weld paths."

Design Constraints

As opposed to defining the behaviors of the system, this third class of requirements typically imposes limitations on the design of the system or process we use to build the system. We'll define a design constraint as:

> *. . .a restriction upon the design of a system, or the process by which a system is developed, that does not affect the external behavior of the system, but must be fulfilled to meet technical, business, or contractual obligations.*

A typical design constraint might be expressed as "Program the welder control unit in Java." In general, we should treat any reasonable design constraints just like any other requirements, although testing compliance to such constraints may require different techniques. Just like functional and nonfunctional requirements, these constraints can play an integral role in designing and testing the system.

Hierarchical Requirements

Many projects benefit from expressing requirements in a hierarchical or parent-child structure. A parent-child requirement amplifies the specificity expressed in a parent requirement. Parent-child requirements give us both a flexible way to enhance and augment a specification, and a means to organize levels of detail. The parent, or top-level specification, is easily understandable to all users; implementers can inspect the more detailed "child" specification to make sure that they understand all of the implementation details. Note that hierarchical requirements consist of the standard three types of requirements: functional, non-functional, and design constraints. The hierarchical approach simply defines the *elaboration relationship* among requirements.

Traceability

In addition to defining the terms we use for things that describe system requirements, we should now turn to a key *relationship*, traceability, which may exist among these things.

A significant factor in quality software is the ability to understand, or trace, requirements through the stages of specification, architecture, design, implementation, and test. Historical data shows that the impact of change is often missed, and small changes to a system can create

significant reliability problems. Therefore, the ability to track relationships, and relate these relationships when change occurs, is key in software quality assurance processes. This is particularly the case for mission critical activities, including safety-critical systems (medical and transportation products), systems with high economic costs of failure (online trading), and so on. Here's how we define requirements traceability:

> *A traceability relationship is a dependency in which the entity (feature, use case, requirement) "traced to" is in some way dependent on the entity it is "traced from."*

For example, we've described how one or more Software Requirements are created to support a given feature or use case specified in the Vision document. Therefore, we can say that these Software Requirements have a *traceability relationship* with one or more Features.

Impact Analysis and Suspectness

A traceability relationship goes beyond a simple dependency relationship because it provides the ability to do impact analysis using a concept that we call "suspectness." A traceability relationship goes "suspect" whenever a change occurs in the "traced from" (independent) requirement, and therefore the "traced to" (dependent requirement) must be checked to ensure that it remains consistent with the requirement from which it is traced.

For example, if we use traceability to relate requirements to specific tests, and if a requirement such as "The Smartbot shall be able to store and retrieve a maximum of 100 weld paths" becomes "The Smartbot shall be able to store *and* retrieve a maximum of 200 weld paths," then the test traced from this requirement is suspect. It is unlikely any test devised for the first requirement will be adequate to test the second one.

Change Requests and the Change Management System

Finally, change is inevitable. For a project to have any hope of succeeding, a process for managing all changes -- including requests that affect features and requirements -- in an orderly manner is essential. The key element of any change management system is the Change Request itself. We'll define a Change Request as:

> *. . .an official request to make a revision or addition to the features and/or requirements of a system.*

Change Requests need to enter the system as structured, formalized statements of proposed changes and any particulars surrounding those changes. In order to manage these changes, it's important that each one have its own identity in the system. A simple Change Request form might look something like this:

Change Request	
Change Request Item	**Value**
Change Request ID	CR001
Change Request Name	Safety Feature on "Power On" Button
Brief Description of Change	Add hold time to "Power On" button that requires user to hold button for xx seconds before system turns on.
Requested by...	Safety Supervisor

A change management system should be used to capture *all* inputs and transmit them to the authority of a Change Control Board (CCB) for resolution. The CCB should consist of no more than three to five people who represent the key stakeholders for the project (customers, marketing, and project management). They administer and control changes to the system, and thereby play a key role in helping the project succeed.

Summary

At the beginning, we noted that a goal of this article was to help practitioners in the field improve their ability to answer the fundamental question:

> *"What, exactly, is this system supposed to do?"*

As a step toward this goal, we defined and described some of the common terms -- such as stakeholder needs, features, use cases, software requirements, and more -- used by analysts and others who have responsibility for describing issues in the problem domain, and for expressing the requirements to be imposed upon any prospective solution. In so doing, we also illustrated some of the key concepts of effective requirements management. By using the terms and approaches outlined in this article, you can better understand your user's needs and communicate requirements for proposed solutions to developers, testers, and other technical team members.

The Unified Modeling Language (UML) is an important technique for further defining and communicating additional key aspects of software solutions. A language for visualizing, specifying, and documenting the artifacts of a software-intensive system, it provides a means for expressing these technical constructs in a more semantically precise manner. The *User Guide* and *Reference Manual* for UML listed below provide practical advice on its use.

Suggested Reading

Leffingwell, Dean, and Don Widrig. *Managing Software Requirements: A Unified Approach*. Reading, MA: Addison Wesley Longman, 1999.

Weigers, Karl. *Software Requirements*. Redmond, WA: Microsoft Press, 1999.

Bibliography

Booch, Grady. *Object-Oriented Analysis and Design with Applications*, 2nd ed. Redwood City, CA: Benjamin Cummings, 1994.

Booch, Grady, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Reading, MA: Addison Wesley Longman, 1999.

Dorfmann, Merlin, and Richard H. Thayer. *Standards, Guidelines, and Examples of System and Software Requirements Engineering*. Los Alamitos, CA: IEEE Computer Society Press, 1990.

Jacobson, Ivar, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Harlow, Essex, England: Addison Wesley Longman, 1992.

Rumbaugh, James, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison Wesley Longman, 1999.

---

[1]At Rational Software, it has been my privilege to work with some of the industry's leading methodologists -- Grady Booch, Ivar Jacobson, Jim Rumbaugh, Philippe Kruchten, Bran Selic, and others. While this has been a rewarding and fascinating part of my career, it's not something I could recommend for everybody. In other words, please do NOT try this at home.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!

Fault Tolerance Techniques for Distributed Systems

by **Bran Selic**
Principal Engineer
Rational Software Canada

As our high-tech society becomes increasingly dependent on computers, the demand for more dependable software will increase and likely become the norm. In the past, fault-tolerant computing was the exclusive domain of very specialized organizations such as telecom companies and financial institutions. With business-to-business transactions taking place over the Internet, however, we are interested not only in making sure that things work as intended, but also, when the inevitable failures do occur, that the damage is minimal. (None of us would be happy to lose money because a fault occurred during the transfer of funds from one account to another, for instance.)

Unfortunately, fault-tolerant computing is extremely hard, involving intricate algorithms for coping with the inherent complexity of the physical world. As it turns out, that world conspires against us and is constructed in such a way that, generally, it is simply not possible to devise absolutely foolproof, 100% reliable software[1]. No matter how hard we try, there is always a possibility that something can go wrong. The best we can do is to reduce the probability of failure to an "acceptable" level. Unfortunately, the more we strive to reduce this probability, the higher the cost.

The Concepts Behind Fault-Tolerant Computing

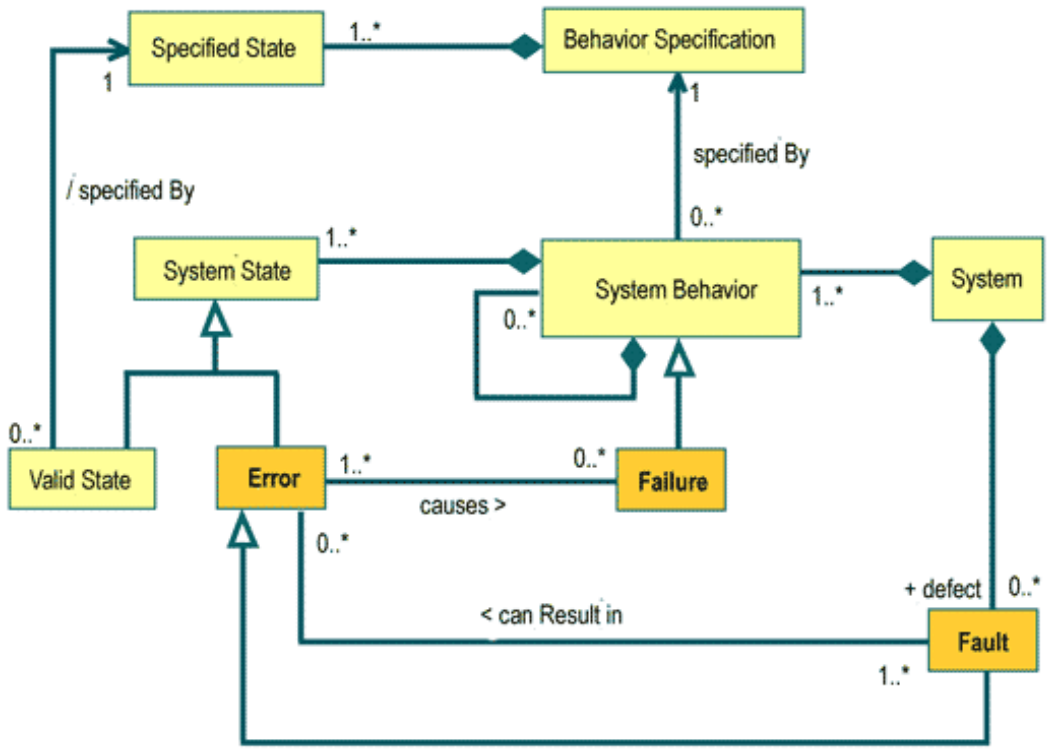
There is much confusion around the terminology used with fault tolerance. For example, the terms "reliability" and "availability" are often used interchangeably, but do they always mean the same thing? What about "faults" and "errors"? In this section, we introduce the basic concepts behind fault tolerance[2] .

Fault tolerance is the ability of a system to perform its function correctly even in the presence of internal faults. The purpose of fault tolerance is to increase the dependability of a system. A complementary but separate approach to increasing dependability is *fault prevention*. This consists of techniques, such as inspection, whose intent is to eliminate the circumstances by which faults arise.

Failures, Errors, and Faults

Implicit in the definition of fault tolerance is the assumption that there is a specification of what constitutes correct behavior. A *failure* occurs when an actual running system deviates from this specified behavior. The cause of a failure is called an *error*. An error represents an invalid system state, one that is not allowed by the system behavior specification. The error itself is the result of a defect in the system or fault. In other words, a *fault* is the root cause of a failure. That means that an error is merely the symptom of a fault. A fault may not necessarily result in an error, but the same fault may result in multiple errors. Similarly, a single error may lead to multiple failures. These basic concepts are illustrated using the Unified Modeling Language (UML) class diagram in Figure 1.

Figure 1: Failures, Errors, and Faults



For example, in a software system, an incorrectly written instruction in a program may decrement an internal variable instead of incrementing it. Clearly, if this statement is executed, it will result in the incorrect value being written. If other program statements then use this value, the whole system will deviate from its desired behavior. In this case, the erroneous statement is the fault, the invalid value is the error, and the failure is the behavior that results from the error. Note that if the variable is never read after being written, no failure will occur. Or, if the invalid statement is never executed, the fault will not lead to an error. Thus, the mere presence of errors or faults does not necessarily imply system failure.

As this example illustrates, the designation of what constitutes a fault -- the underlying cause of a failure -- is relative in the sense that it is simply a point beyond which we do not choose to delve further. After all, the

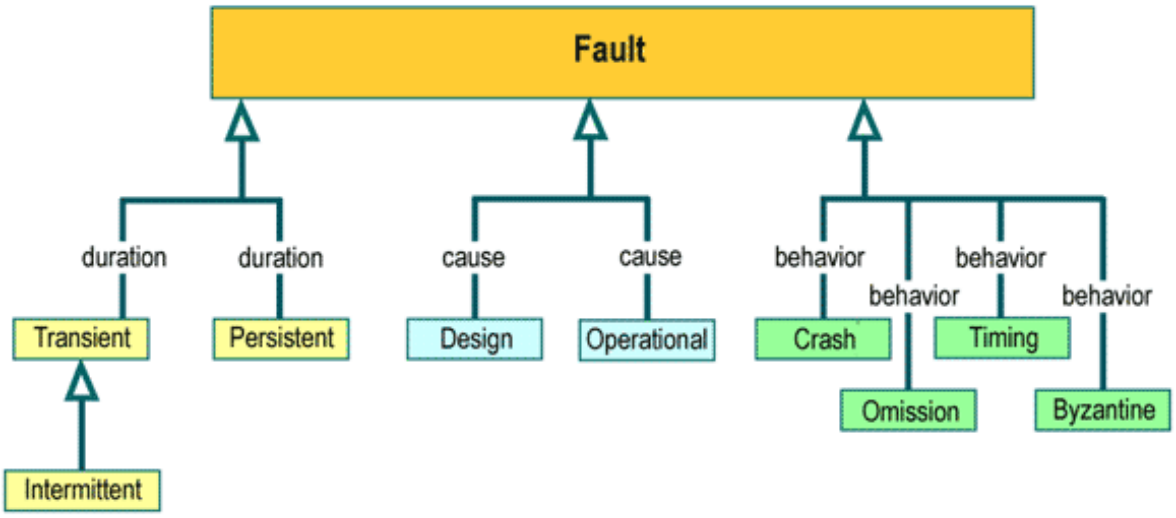
incorrect statement itself is really an error that arose in the process of writing the software, and so on.

At the heart of all fault tolerance techniques is some form of *masking redundancy*. This means that components that are prone to defects are replicated in such a way that if a component fails, one or more of the non-failed replicas will continue to provide service with no appreciable disruption. There are many variations on this basic theme.

Fault Classifications

It is helpful to classify faults in a number of different ways, as shown by the UML class diagram in Figure 2.

Figure 2: Different Classifications of Faults



Based on *duration*, faults can be classified as *transient* or *permanent*. A transient fault will eventually disappear without any apparent intervention, whereas a permanent one will remain unless it is removed by some external agency. While it may seem that permanent faults are more severe, from an engineering perspective, they are much easier to diagnose and handle. A particularly problematic type of transient fault is the *intermittent* fault that recurs, often unpredictably.

A different way to classify faults is by their underlying *cause*. *Design faults* are the result of design failures, like our coding example above. While it may appear that in a carefully designed system all such faults should be eliminated through fault prevention, this is usually not realistic in practice. For this reason, many fault-tolerant systems are built with the assumption that design faults are inevitable, and theta mechanisms need to be put in place to protect the system against them. *Operational faults*, on the other hand, are faults that occur during the lifetime of the system and are invariably due to physical causes, such as processor failures or disk crashes.

Finally, based on how a failed component behaves once it has failed, faults can be classified into the following categories:

- *Crash faults* -- the component either completely stops operating or never returns to a valid state;
- *Omission faults* -- the component completely fails to perform its service;
- *Timing faults* -- the component does not complete its service on time;
- *Byzantine faults* -- these are faults of an arbitrary nature.[3]

General Fault Tolerance Procedure

In general, the process for dealing with faults can be grouped into a series of distinct activities that are typically (although not necessarily) performed in sequence, as shown in the UML activity diagram in Figure 3.

Error detection is the process of identifying that the system is in an invalid state. This means that some component in the system has failed. To ensure that the effects of the error are limited, it is necessary to isolate the failed component so that its effects are not propagated further. This is known as *damage confinement*. In the error recovery phase, the error and -- more importantly -- its effects, are removed by restoring the system to a valid state. Finally, in *fault treatment*, we go after the fault that caused the error so that it can be isolated. In other words, we first treat the symptoms and then go after the underlying cause. While it may seem more appropriate to go after the fault immediately, this is often not practical, since diagnosing the true cause of an error can be a very complex and lengthy process. In a software system, it is typical for a single fault to cause many cascading errors that are reported independently. Correlating and tracing through a potential multitude of such error reports often requires sophisticated reasoning.

Error Detection

The most common techniques for error detection are:

- *Replication checks* -- In this case, multiple replicas of a component perform the same service simultaneously. The outputs of the replicas are compared, and any discrepancy is an indication of an error in one or more components. A particular form of this that is often used in hardware is called *triple-modular redundancy* (TMR), in which the output of three independent components is compared, and the output of the majority of the components is actually passed on[4]. In software, this can be achieved by providing multiple independently developed realizations of the same component. This is called *N-version programming*.

- *Timing checks* -- This is used for detecting timing faults. Typically a timer is started, set to expire at a point at which a given service is expected to be complete. If the service terminates successfully before the timer expires, the timer is cancelled. However, if the timer times out, then a timing error has occurred. The problem with timers is in cases where there is variation in the execution of a function. In such cases, it is dangerous to set the timer too tightly, since it may indicate false positives. However, setting it too loosely would delay the detection of the error, allowing the effects to be propagated much more widely.

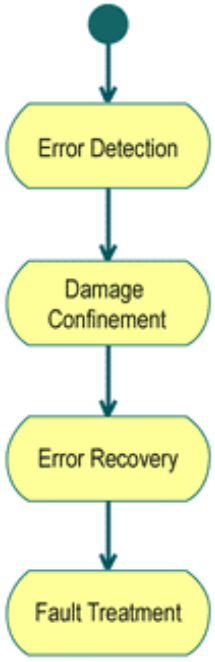


Figure 3: General Fault Tolerance Procedure

- *Run-time constraints checking* -- This involves detecting that certain constraints, such as boundary values of variables not being exceeded, are checked at run time. The problem is that such checks introduce both code and performance overhead. A particular form is *robust data structures*, which have built-in redundancy (e.g., a checksum). Every time these data structures are modified, the redundancy checks are performed to detect any inconsistencies. Some programming languages also support an *assertion* mechanism.
- *Diagnostic checks* -- These are typically background audits that determine whether a component is functioning correctly. In many cases, the diagnostic consists of driving a component with a known input for which the correct output is also known.

Damage Confinement

This consists of first determining the extent to which an error has spread (because time may have passed between the time the error occurred and the time it is detected). It requires understanding the flow of information in a system and following it -- starting from a known failed component. Each component along such flows is checked for errors, and the boundary is determined. Once this boundary is known, that part of the system is isolated until it can be fixed.

To assist with damage confinement, special "firewalls" are often constructed between components. This implies a loose coupling between components so that components outside of the firewall can be easily de-coupled from the faulty components inside and re-coupled to an alternative, error-free redundant set. The cost of this is system overhead, both in performance (communication through a firewall is typically slower) and memory.

Error Recovery

To recover from an error, the system needs to be restored to a valid state.

There are two general approaches to achieving this. In *backward error recovery*, the system is restored to a previous known valid state. This often requires *checkpointing* the system state and, once an error is detected, rolling back the system state to the last checkpointed state. Clearly, this can be a very expensive capability. Not only is it necessary to keep copies of previous states, but also it is necessary to stop the operation of the system during checkpointing to ensure that the state that is stored is consistent. In many cases, this is not viable, since it may not be possible to restore the environment in which the system is operating to the state that corresponds to the checkpointed state.

For such cases, *forward error recovery* is more appropriate. This involves driving the system from the erroneous state to a new valid state. It may be difficult to do unless the fault that caused the error is known precisely and is well isolated, so that it does not keep interfering. Because it is tricky, forward error recovery is not used often in practice.

Fault Treatment

In this phase, the fault is first isolated and then repaired. The repair procedure depends on the type of fault. Permanent faults require that the failed component be replaced by a non-failed component. This requires a *standby* component. The standby component has to be integrated into the system, which means that its state has to be synchronized with the state of the rest of the system. There are three general types of standby schemes:

- *Cold standby* -- This means that the standby component is not operational, so that its state needs to be changed fully when the cutover occurs. This may be a very expensive and lengthy operation. For instance, a large database may have to be fully reconstructed (e.g., using a log of transactions) on a standby disc. The advantage of cold standby schemes is that they do not introduce overhead during the normal operation of the system. However, the cost is paid in fault recovery time.
- *Warm standby* -- In this case, the standby component is used to keep the last checkpoint of the operational component that it is backing up. When the principal component fails, the backward error recovery can be relatively short. The cost of warm standby schemes is the cost of backward recovery discussed earlier (mainly high overhead).
- *Hot standby* -- In this approach, the standby component is fully active and duplicating the function of the primary component. Thus, if an error occurs, recovery can be practically instantaneous. The problem with this scheme is that it is difficult to keep two components in lock step. In contrast to warm standby schemes, in which synchronization is only performed during checkpoints, in this case it has to be done on a constant basis. Invariably, this requires communications between the primary and the standby, so that the overhead of these schemes is often higher than the overhead for warm standby.

Dependability

Dependability means that our system can be trusted to perform the service for which it has been designed. Dependability can be decomposed into specific aspects. *Reliability* characterizes the ability of a system to perform its service correctly when asked to do so. *Availability* means that the system is available to perform this service when it is asked to do so. *Safety* is a characteristic that quantifies the ability to avoid catastrophic failures that might involve risk to human life or excessive costs. Finally, *security* is the ability of a system to prevent unauthorized access.

Technically, reliability is defined as the probability that a system will perform correctly up to a given point in time. A common measure of reliability, therefore, is the *mean time between failures* (MTBF).

Availability is defined as the probability that a system is operational at a given point in time. For a given system, this characteristic is strongly dependent on the time it takes to restore it to service once a failure occurs. A common way of characterizing this is *mean time to repair* (MTTR).

The two measures for reliability (MTBF) and availability (MTBR) can be used to show the relationship between these two important measures. It is important to distinguish these two technical terms, since they are often used interchangeably in everyday communications. This can lead to confusion. The availability of a system can be calculated from these two measures according to the formula:

Availability = (MTBF) / (MTTR + MTBF)

Note that for systems that never fail, availability is equal to reliability.

Distributed Systems

We define a distributed software system (Figure 4) as: *a system with two or more independent processing sites that communicate with each other over a medium whose transmission delays may exceed the time between successive state changes.*

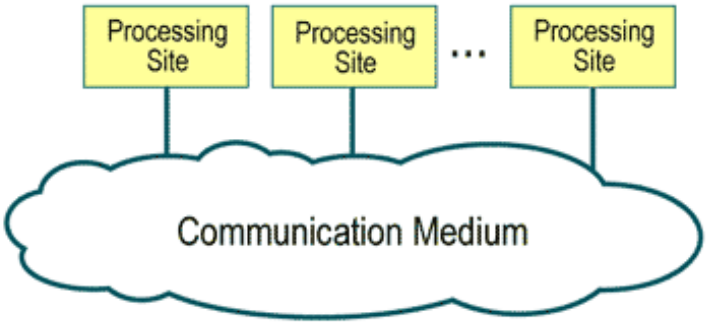


Figure 4: A Distributed System

From a fault-tolerance perspective, distributed systems have a major advantage: They can easily be made redundant, which, as we have seen, is

at the core of all fault-tolerance techniques. Unfortunately, distribution also means that the imperfect and fault-prone physical world cannot be ignored, so that as much as they help in supporting fault-tolerance, distributed systems may also be the source of many failures. In this section we briefly review these problems.

Processing Site Failures

The fact that the processing sites of a distributed system are independent of each other means that they are independent points of failure. While this is an advantage from the viewpoint of the user of the system, it presents a complex problem for developers. In a centralized system, the failure of a processing site implies the failure of all the software as well. In contrast, in a fault-tolerant distributed system, a *processing site failure* means that the software on the remaining sites needs to detect and handle that failure in some way. This may involve redistributing the functionality from the failed site to other, operational, sites, or it may mean switching to some emergency mode of operation.

Communication Media Failures

Another kind of failure that is inherent in most distributed systems comes from the communication medium. The most obvious, of course, is a permanent hard failure of the entire medium, which makes communication between processing sites impossible. In the most severe cases, this type of failure can lead to partitioning of the system into multiple parts that are completely isolated from each other. The danger here is that the different parts will undertake activities that conflict with each other.

A different type of media failure is an intermittent failure. These are failures whereby messages travelling through a communication medium are lost, reordered, or duplicated. Note that these are not always due to hardware failures. For example, a message may be lost because the system may have temporarily run out of memory for buffering it. Message reordering may occur due to successive messages taking different paths through the communication medium. If the delays incurred on these paths are different, they may overtake each other. Duplication can occur in a number of ways. For instance, it may result from a retransmission due to an erroneous conclusion that the original message was lost in transit.

One of the central problems with unreliable communication media is that it is not always possible to positively ascertain that a message that was sent has actually been received by the intended remote destination. A common technique for dealing with this is to use some type of *positive acknowledgement protocol.* In such protocols, the receiver notifies the sender when it receives a message. Of course, there is the possibility that the acknowledgement message itself will be lost, so that such protocols are merely an optimization and not a solution.

The most common technique for detecting lost messages is based on time-outs. If we do not get a positive acknowledgement within some reasonable time interval that our message was received, we conclude that it was dropped somewhere along the way. The difficulty of this approach is to

distinguish situations in which a message (or its acknowledgement) is simply slow from those in which a message has actually been lost. If we make the time-out interval too short, then we risk duplicating messages and also reordering in some cases. If we make the interval too long, then the system becomes unresponsive.

Transmission Delays

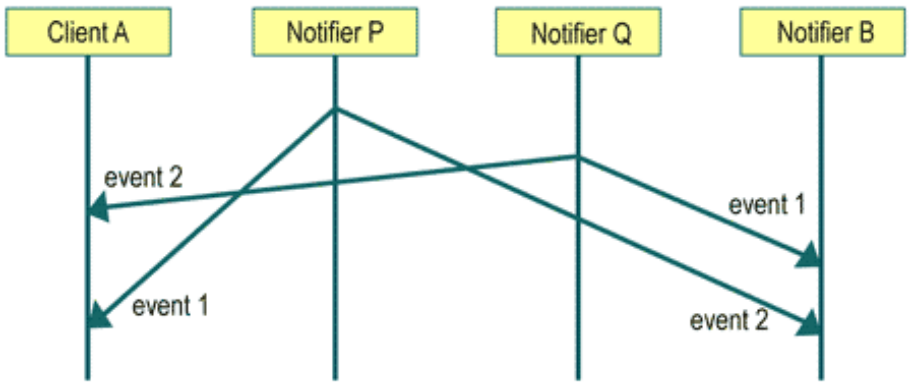
While transmission delays are not necessarily failures, they can certainly lead to failures. We've already noted that a delay can be misconstrued as a message loss.

There are two different types of problems caused by message delays. One type results from *variable* delays (jitter). That is, the time it takes for a message to reach its destination may vary significantly. The delays depend on a number of factors, such as the route taken through the communication medium, congestion in the medium, congestion at the processing sites (e.g., a busy receiver), intermittent hardware failures, etc. If the transmission delay is constant, then we can much more easily assess when a message has been lost. For this reason, some communication networks are designed as synchronous networks, so that delay values are fixed and known in advance.

However, even if the transmission delay is constant, there is still the problem of *out-of-date information*. Since messages are used to convey information about state changes between components of the distributed system, if the delays experienced are greater than the time required to change from one state to the next, the information in these messages will be out of date. This can have major repercussions that can lead to unstable systems. Just imagine trying to drive a car if visual input to the driver were delayed by several seconds.

Transmission delays also lead to a complex situation that we will refer to as the *relativistic effect*. This is a consequence of the fact that transmission delays between different processing sites in a distributed system may be different. As a result, different sites may see the same set of messages but in a different order. This is illustrated in Figure 5 below:

Figure 5: The Relativistic Effect



In this case, distributed sites `NotifierP` and `NotifierQ` each send out a notification about an event to the two clients (`ClientA` and `ClientB`). Due to the different routes taken by the individual messages and the different delays along those routes, we see that `ClientB` sees one sequence (`event1` followed by `event2`), whereas `ClientA` sees a different one (`event2-event1`). As a consequence, the two clients may reach different conclusions about the state of the system.

Note that the mismatch here is not the result of message overtaking (although this effect is compounded if overtaking occurs); it is merely a consequence of the different locations of the distributed agents relative to each other.

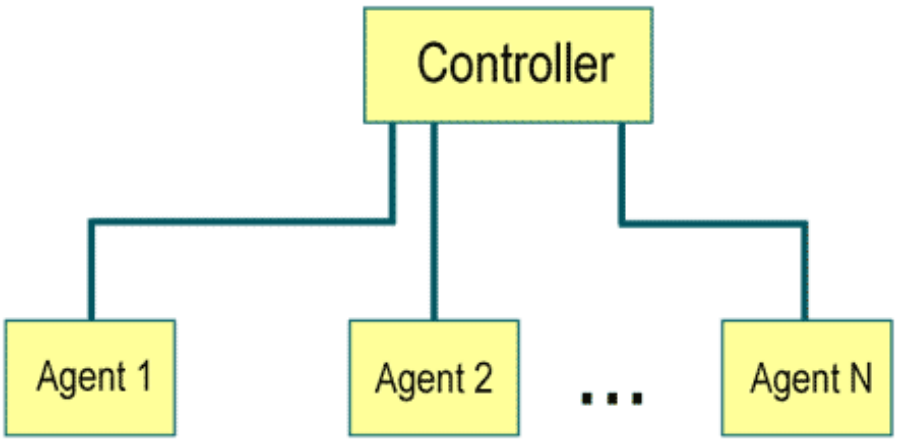
Distributed Agreement Problems

The various failure scenarios in distributed systems and transmission delays in particular have instigated important work on the foundations of distributed software.[5] Much of this work has focused on the central issue of *distributed agreement*. There are many variations of this problem, including time synchronization, consistent distributed state, distributed mutual exclusion, distributed transaction commit, distributed termination, distributed election, etc. However, all of these reduce to the common problem of reaching agreement in a distributed environment in the presence of failures.

A Fault-Tolerant Pattern for Distributed Systems

We now examine a specific pattern that has been successfully used to construct complex, fault-tolerant embedded systems in a distributed environment. This pattern is suitable for a class of distributed applications that is characterized by the star-like topology shown in Figure 6, which commonly occurs in practice.

Figure 6: System Topology



The system consists of a set of distributed agents, each on a separate processing site, which collaboratively perform some function. The role of the controller is to coordinate the operation of the agents. Note that it is not

necessarily the case that the agents have to communicate through the controller. We have simply not shown such connections in our diagram since they are not relevant at this point. We make the following assumptions about this system:

- *Soft response time* -- The processing times for global functions allow for some variability.
- *Non-critical agents* -- The system may still be useful even if one or more agents fail permanently.

In this system, the permanent failure of the central controller would lead to the loss of all global functionality. Hence, it is a single point of failure that needs to be made fault tolerant. However, we would like to avoid the overhead of a full hot standby -- or even a warm standby -- for this component.

The essential feature of this approach is to distribute the state information about the system as a whole between the controller and the agents. This is done in such a way that (a) the controller holds the global state information that comprises the state information for each agent, and (b) each agent keeps its own copy of its state information. Every time the local state of an agent changes, it informs the controller of the change. The controller caches this information. Thus, we have state redundancy. Note that there is no need for a centralized consistent checkpoint of the entire system, which, as we have mentioned, is a complex and high-overhead operation.

Obviously, the state redundancy does not protect us from permanent failures of the controller. Note however, that any local functions performed by the agents are unaffected by the failure of the agents. Global functions, on the other hand, may have to be put on hold until the controller recovers. Thus, it is critical to be able to recover the controller.

Controller recovery can be easily achieved by using a simple, low-overhead, cold standby scheme. The standby controller can recover the global state information of the failed controller simply by querying each agent in turn. Once it has the full set, the system can resume its operation; the only effect is the delay incurred during the recovery of the controller. Of course, it is also possible to use other standby schemes as well with this approach.

If an agent fails and recovers, it can restore its local state information from the controller. One interesting aspect to this topology is that it is not necessary for the controller to monitor its agents. A recovering agent needs merely to contact its controller to get its state information. This eliminates costly polling.

Additional References

Goyer, P., P. Momtahan, and B. Selic. "A Synchronization Service for Locally Distributed Applications," in M. Barton, et al., ed., *Distributed Processing* (North Holland, 1988), pp. 3-17.

Goyer, P., P. Momtahan, and B. Selic. "A Fault-Tolerant Strategy for

Hierarchical Control in Distributed Computer Systems," *Proc. 20th IEEE Symp. on Fault-Tolerant Computing Systems (FTCS20)*, (IEEE CS Press, 1990), pp. 290-297.

Jalote, P. *Fault Tolerance in Distributed Systems*, (Prentice Hall, 1994).

Randell, B., P. Lee, and P. Treleaven. "Reliability in Computing System Design," *ACM Computing Surveys*, Vol. 10, No. 2, June 1978, pp. 123-165.

---

[1] Fischer, M., N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, Vol. 32, No. 2, April 1985, pp. 374-382.

[2] Halpern, J. and Y. Moses, "Knowledge and Common Knowledge in a Distributed Environment," *Proc. of the 3rd ACM Symposium on Principles of Distributed Systems*, 1984, pp. 50-61 and Lamport, L., R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, Vol. 4 No. 3, July 1982, pp. 382-401.

[3] The rather curious name for this category of faults comes from an analogy with the reputed tendency of the officers of ancient Byzantium, who often conspired to overthrow their monarch of the day. A Byzantine failure, therefore, represents the most difficult type of fault: one that is caused by malicious intent. Clearly, a system that can protect itself from Byzantine faults is impervious to any kind of fault.

[4] A major weakness of TMR and similar schemes is that the comparator unit is not redundant, so that any failures of this component are not masked.

[5] See in particular these four sources: 1) Garcia-Molina, H., "Elections in a Distributed Computing System, *IEEE Trans. on Computers*, Vol. C-31, No. 1, January 1982, pp. 48-59. 2) Hadzilacos, V. and S. Toueg, "Fault-Tolerant Broadcasts and Related Problems," Chapter 5 in S. Mullender (ed.), *Distributed Systems*, 2nd ed., Addison-Wesley, 1993, pp. 97-145. 3) Halpern, J. and Y. Moses, "Knowledge and Common Knowledge in a Distributed Environment," *Proc. of the 3rd ACM Symposium on Principles of Distributed Systems*, 1984, pp. 50-61. 4)Lamport, L., R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, Vol. 4 No. 3, July 1982, pp. 382-401.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!

Five Days of the Fish

An Allegory by **Joe Marasco**
Senior Vice President
Rational Software

A dead fish begins to smell bad on the second day after it dies; we say something stinks like three-day-old fish. Business problems are a lot like dead fish. Typically, people perceive them only when they start to stink -- i.e., after they have been incubating for some time. Very often, managers like you and me are asked to jump in and "fix" such problems, either internally or for our customers. Because in most "rotting-fish" situations time is of the essence, I would like to offer some guidance, in allegorical form, to help other "fixers" analyze and resolve these crises.

Let's imagine an exotic fish market, where fish are kept alive in a display tank until just before they are sold, so that they can be eaten when fresh. Needless to say, the store has high prices, high overhead, and low volume, leading to very thin profit margins.

This market has in its employ a small but very vocal group of Cassandras who periodically announce that a fish is about to die, although all the fish are cruising around the tank as usual. This group consists of:

- Employees who don't have enough to do and are always looking for things that are not quite right. Like Nostradamus, they are never taken to task if their predictions don't come true. They therefore predict all kinds of things with great abandon, hitting it right every once in awhile just because of statistical probability. The signal-to-noise ratio for these people is very low, so usually the owner can safely ignore them.
- A few individuals who actually *can* see what's coming and aren't afraid to vocalize about it. Provided that the owner can distinguish these folks from the first subgroup, they can be useful to him, but they are not always right, either.

Day One: Unaware

That very week, a fish *does* die, but very few people take notice. One or two clueless employees poke around and discuss what tests they might perform to decide if it is normal for fish to float on the top of the tank. Later, they will remark that they suspected something was wrong but weren't sure.

Day Two: Avoiding the Issue

The following day, more people perceive that something is wrong, although some speculate that the fish is just tired. There is not yet enough stink in the air to arouse them to action. Although customer traffic has dropped off a bit, the head clerk delays telling the owner, who is notorious for flying into a rage upon receipt of bad news.

Day Three: Enter "The Fixer"

Some time during the third day, when the dead fish really begins to smell, the clerk finally summons the owner, who, in turn, calls in an outside specialist. This "fixer" arrives on the scene and pronounces the fish legally dead. The owner, in complete denial, scolds the fixer for not employing mouth-to-mouth resuscitation. When the fixer points out the futility of such an action, citing the fish's odor, the owner then commands him to "fix the fish problem."

What does this mean? Banishing the odor and/or keeping any more fish in the tank from dying? Working with suppliers to get healthier fish in the first place? Finding someone who wants to buy the dead, putrefying fish? All of the above? Knowing that the owner does not yet know what he wants -- and that his idea of "fix" will probably change a few times before the fixer is done -- the fixer rolls up his sleeves and gets to work. In the meantime, business continues to fall off.

No one can ignore the odor, and customers are staying away. Rumors spread throughout the store that many fish have died and were surreptitiously thrown away. The Cassandras are abuzz with "I told you so," and employees are putting as much distance as possible between themselves and the dead fish. The owner assures everyone that the fixer is working on the problem, and will soon have a solution.

Meanwhile, the fixer is up to his elbows in stinking fish guts, feeling dirty, lonely, and desperate. The owner tells him what a prince he is. Aware that this is not the first time he has been called in on a "dead fish" problem, the fixer is mildly upset and concerned about the extent to which he is being "managed," but knows that to be successful he must keep his head down and solve the problem.

Day Four: The Turning Point

By day four, the stench is so bad that the employees are clamoring for the

fixer's head. Some even blame him for the fish's death. Moreover, the health department arrives and threatens to close the store, sending the owner into a full panic.

The fixer is prepared for all of this. He knows that day four is his moment of truth -- the get-rolling-or-smell-horrible-forever day. That is why he wasted no time trying to resuscitate the dead fish on day three, knowing from experience that he would need time to examine the entrails and formulate a solution. Today, he can legitimately announce the solution, begin implementation, and start calming everyone down.

He makes no attempt to control the odor (focus on PR) yet, understanding that doing so before a true solution is underway would invite catastrophe. Privately, the fixer thanks his lucky stars that the remaining fish appear to be hanging in there, because another dead fish at this point could push the whole thing over the cliff.

Day Five: Two Critical Paths

Day five is as critical as day four. It has been a long time since the fish died. Employees cannot bear the smell any longer, and the fixer knows that if it doesn't disappear quickly, things will take a dramatic turn for the worse. So he divides his efforts between two tasks: resolving the problem and getting rid of the odor. The fixer must balance the two tasks carefully. If he neglects his problem-solving work, more fish may die. If he ignores the odor, the owner might mistakenly interpret the ongoing stench as a lack of progress and jettison the fixer's proposed solution -- tossing the baby out with the bathwater, so to speak.

In addition to working on these tasks, the fixer also urges the owner to offer the remaining fish at half price, even though there is absolutely nothing wrong with them. With this incentive, many new customers drop in and purchase fish, despite the lingering odor. These people leave feeling self-satisfied with their business and culinary acumen. Some established customers, however, adopt a wait-and-see attitude, wanting to be sure that the rest of the fish are not diseased before they eat one.

If the fixer can survive day five, things will get better. Both employees and customers will quickly forget about the problem, and he can finish resolving it without all the pressure. He will succeed by making steady, regular progress and ensuring that no new fish die on his watch. Until the problem is fully resolved, ongoing vigilance is critical.

Moral of the Story

Fixers of the world, remember that in a crisis you have three days to prove yourself:

- Fish start to stink on day two, but you will rarely be called that early.
- Don't waste day three fooling around. Open up the fish and conceive a plan. The stink will be bad, and it will get worse.
- Announce your plan and begin executing it on day four.
- Devote day five to working on the problem *and* eliminating the odor.

Good luck! Work on keeping your fish alive, and may all your dead fish be little ones.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!